Model-Based Product-Line Regression Testing of Variants and Versions of Variants

Von der

Carl-Friedrich-Gauß-Fakultät

der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines

Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von

Sascha Burkhard Lity

geboren am 28.09.1985

in Bad Pyrmont

Eingereicht am:	03.06.2019
Disputation am:	22.07.2019
1. Referentin:	Prof. Dr.-Ing. Ina Schaefer
2. Referent:	Prof. Dr. rer. nat. Andy Schürr

2019

Abstract

The increasing complexity and prevalence of software-intensive systems encourage the development of variant-rich systems such as software product lines (SPL). Although the development of SPLs is well understood, their quality assurance achieved via testing remains an open field of research. Software testing is a crucial and challenging activity of the software development process and gets even more challenging in the context of SPLs. In general, the application of single-software testing techniques for SPL testing is not practical as it leads to the individual testing of a potentially vast number of variants. In addition, testing each variant in isolation results in redundant testing processes by means of redundant test-case executions due to the shared commonality. Existing techniques for SPL testing cope with those challenges (1) by identifying samples of variants to be tested, (2) by prioritizing the set of variants under test, or (3) by creating reusable test artifacts for the SPL test process. However, each variant is still tested separately without taking the explicit knowledge about the shared commonality and variability into account to reduce the overall testing effort. Furthermore, due to the increasing longevity of software systems, their development has to face software evolution. Hence, quality assurance has also to be ensured after SPL evolution by testing respective versions of variants. To exploit the commonality and reuse potential of test artifacts during testing of evolving SPLs, the adoption of regression testing strategies, e.g., retest test selection facilitates the incremental testing of variants and versions of variants by focusing on the differences, i.e., variability, between them such that an effort reduction can be achieved.

In this thesis, we tackle the challenges of testing redundancy as well as evolution by proposing a framework for model-based regression testing of evolving SPLs. The framework facilitates efficient incremental testing of variants and versions of variants by exploiting the commonality and reuse potential of test artifacts and test results. For the realization of the framework, our contribution is divided into three parts. First, we propose a test-modeling formalism capturing the variability and version information of evolving SPLs in an integrated fashion. The formalism builds the basis for automatic derivation of reusable test cases as well as for the efficient application of change impact analysis to guide retest test selection. Second, we introduce two techniques for incremental change impact analysis to identify (1) changing execution dependencies to be retested between subsequently tested variants and versions of variants, and (2) the impact of an evolution step by means of changes to the variant set when testing the next SPL version in terms of modified, new as well as unchanged versions of variants. Third, we define a coverage-driven retest test selection based on a new retest coverage criterion that incorporates the results of the change impact analysis. The retest test selection facilitates the reduction of redundantly executed test cases during incremental testing of variants and versions of variants. The framework is prototypically implemented and evaluated by means of three evolving SPLs showing that it achieves a reduction of the overall effort for testing evolving SPLs and, therefore, enables a more targeted use of the limited test resources.

Zusammenfassung

In der heutigen Zeit wird immer mehr Funktionalität durch Software bereitgestellt und ist somit aus unserem alltäglichen Leben nicht mehr wegzudenken. Die dabei steigende Komplexität und die Einbeziehung von Kundenwünschen in die Entwicklung von Software-Systemen führt allerdings dazu, dass es problematischer wird, einzelne Systeme jeweils komplett neu zu realisieren. Aus diesen und auch anderen Gründen ist ein Trendwechsel in der Sofwareentwicklung zu erkennen, bei dem sich der Fokus auf die Entwicklung von variantenreichen Systemen wie zum Beispiel Softwareproduktlinen verschiebt. Softwareproduktlinen werden bereits in der Industrie in vielen Bereichen eingesetzt, so dass deren Entwicklung wohlbekannt ist. Allerdings ist das effiziente Testen von Softwareproduktlinen und somit deren Qualitätssicherung immernoch ein offenes Problem.

Das Testen von Software ist ein wichtiger und ebenso anspruchsvoller Bestandteil der Softwareentwicklung und wird für Softwareproduktlinien durch die Berücksichtigung von Variabilität noch herausfordernder. Die Anwendung von Techniken, die für das Testen einzelner Systeme entwickelt wurden, ist jedoch im Allgemeinen nicht praktikabel. Zum einen ist die Anzahl an potentiell zu testenden Varianten zu umfangreich, um sie einzeln zu testen. Zum anderen resultiert der unabhängige Test jeder Variante in redundanten Testfallausführungen, die durch die Gemeinsamkeiten zwischen Varianten hervorgerufen werden. Existierende Ansätze adressieren die Herausforderungen beim Testen von Softwareproduktlinien indem sie (1) die Anzahl an zu testenden Varianten reduzieren, (2) eine Reihenfolge für das Testen der Variantenmenge bestimmen oder (3) wiederverwendbare Testartefakte wie zum Beispiel Testfälle für den Testprozess automatisch erstellen. Das Problem des redundanten Testens besteht allerdings auch bei diesen Ansätzen weiterhin, da jede Variante unabhängig getestet wird, ohne dabei das explizite Wissen über Gemeinsamkeiten und Variabilität auszunutzen, um den Testaufwand zu reduzieren. Neben diesen Schwierigkeiten muss sich die Entwicklung als auch die Qualitätssicherung mit der Langlebigkeit und somit der Evolution von Software auseinandersetzen. Dies birgt weitere Herausforderungen für das Testen von Softwareproduktlinien, da nicht nur für Varianten sondern auch für ihre Versionen die Qualität sichergestellt werden muss. Um die Gemeinsamkeiten und das Wiederverwendungspotential von Testartefakten beim Testen von evolvierenden Softwareproduktlinien auszunutzen, können Strategien des Regressionstesten für das inkrementelle Testen von Varianten und Variantenversionen angewendet werden. Das Regressionstesten fokussiert sich dabei auf die Unterschiede und somit die Variabilität zwischen Varianten und Variantenversionen, um eine Testaufwandsreduzierung zu ermöglichen, indem zum Beispiel Testfälle für einen Wiederholungstest selektiert werden anstatt die komplette Testfallmenge erneut auszuführen.

In dieser Arbeit stellen wir ein Framework für das modellbasierte Regressionstesten von evolvierenden Softwareproduktlinien vor, durch das wir die Herausforderungen des redundanten Testens sowie der Qualitätssicherung nach Evolution adressieren. Unser Framework ermöglicht einen inkrementellen Testprozess um Varianten und Variantenversionen effizient zu testen. Hierbei nutzen wir das explizite Wissen über gemeinsame Funktionalität sowie das Wiederverwendungspotential

von Testartefakten und Testresultaten aus. Für die Realisierung des Frameworks präsentiert diese Arbeit drei Beiträge. Als ersten Beitrag definieren wir einen neuen Ansatz zur Testmodellierung, der die Variabilitäts- sowie Versionsinformation von evolvierenden Softwareproduktlinien gleichermaßen in den Modellierungsprozess mit einbezieht und somit erfasst. Der Testmodellierungsansatz dient als Grundlage, um zum einen die automatische Generierung von wiederverwendbaren Testfällen und zum anderen die automatische Selektierung von Testfällen durch die effiziente Anwendung von Änderungsauswirkungsanalysen zu ermöglichen. Als zweiten Beitrag entwickeln wir zwei neue Techniken für die Änderungsauswirkungsanalyse. Die erste Technik ermöglicht die Identifikation von Änderungen in Ausführungsabhängigkeiten zwischen nacheinander zu testenden Varianten und Variantenversionen, die erneut durch einen Wiederholungstest überprüft werden sollten. Die zweite Technik bestimmt die Auswirkungen eines Evolutionsschrittes auf die Menge der Varianten der als nächstes zu testenden Version einer Softwareproduktlinie. Die Auswirkungen auf die Menge der Varianten werden dabei durch die Kategorisierung in modifizierte, neue oder unveränderte Variantenversionen klassifiziert. Als dritten Beitrag definieren wir eine abdeckungsgetriebene Selektion von Testfällen, um einen Wiederholungstest durchführen zu können. Hierfür schlagen wir ein neues Abdeckungskriterium vor, das die Resultate der Änderungsauswirkungsanalyse einbezieht, um Entscheidungen für einen Wiederholungstest automatisiert treffen zu können. Das Selektionsverfahren ermöglicht somit die Reduktion der redundanten Testfallausführungen während des inkrementellen Testens von Varianten und Variantenversionen. Das Framework ist prototypisch implementiert und wurde anhand von drei evolvierenden Softwareproduktlinien evaluiert. Die Resultate zeigen das eine Reduktion des Gesamtaufwands für das Testen evolvierender Softwareproduktlinien erreicht und somit ein gezielterer Einsatz der limitierten Testressourcen ermöglicht wird.

Acknowledgements

During the time of my PhD, many people have accompanied, influenced, and supported me in different ways. In the following, I would like to sincerely thank the people which are the most related to this thesis.

As the first person, I want to thank my supervisor Ina Schaefer for giving me the opportunity to finish my PhD at her institute. I am very grateful that she took over the mentoring and included me in her team. She has always given me great advice and found the time for discussions independent whether it was just an idea or a rather difficult part of my thesis. I also would like to thank Ursula Goltz the former head of the Institute for Programming and Reactive Systems that she gave me the chance to start my PhD after my master's degree. I thank my reviewer Andy Schürr for taking the effort to review my thesis as well as for the successful cooperation in the IMoTeP research project.

I would like to thank my colleagues at TU Braunschweig both from the Institute for Programming and Reactive Systems and the Institute of Software Engineering and Automotive Informatics for the great working atmosphere and the strong team spirit. It was a pleasure to be a part of both teams. In particular, I would like to thank Thomas Thüm, who supported and improved my research and, therefore, this thesis based on (most of the time) long discussions, his advices, and his feedback. I will also not forget that you presented our paper at ICSR 2016 such that I could meanwhile marry my wife and had an accepted paper to be integrated in my thesis. Another big thanks goes to Stephan Mennicke. Our morning coffee sessions, where we discussed our current research topics and problems during the process of writing our theses, have always been a enjoyable start of the day and I will definitely miss them. I want to thank Malte Lochau, Johannes Bürdek, Lars Luthmann, and Mustafa Al-Hajjaji for our collaborations and fruitful discussions which have also influenced this thesis. In addition, I would like to thank Sabrina Lischke, Tabea Bordis, Hauke Baller, and Remo Lachmann, who shared an office with me during the time of my PhD and who always had an open ear and time for discussing new ideas.

Over the years, I also received support by many students which I supervised. Special thanks go to Sophia Nahrendorf, Manuel Nieke, and Nils-André Fohrjan, who helped me as student researchers to implement and evaluate many of the contributions of this thesis.

I would like to thank my family for being there for me in times of need, for their support, and guidance. My parents always encouraged me to follow my interests and enabled me to pursue this educational path. Many thanks also go to my friends from Braunschweig and Emmerthal. You helped me to think of more than just work from time to time.

Finally, I thank my beloved wife Anja and my son Mirko for their love and support in all situations. You have always given me motivation and strength to accomplish this achievement. Thank you so much. This success also belongs to you.

Contents

1 Introduction

Software has become an integral part of our everyday life. Its application ranges from simple systems, e.g., calculators, to complex software-intensive systems, e.g., facilitating the control of (sub)systems in safety-critical domains such as automotive, rail, or avionic. In addition, more and more functionality is realizable by software such that the prevalence of software-intensive systems increases. As a result of this increasing prevalence as well as the demand for customer-individualized software systems, single-software development reaches its limits [CN01; PBvdL05; LSR07]. Therefore, the development trend shifts from the realization of customer-specific systems from scratch to the mass customization of software systems. Mass customization facilitates the mass production of customer-individualized software based on the large-scale reuse of development artifacts [PBvdL05; LSR07]. A prominent paradigm to accomplish mass customization is the development of software product lines (SPL). An SPL defines a family of similar software variants applied in the same domain, where further the commonality and variability between those variants is explicitly documented by means of features [PBvdL05], i.e., customer-visible functionality [KCH+90]. However, although the development of SPLs is well understood [SD07; CAA09; SRC+12; ABK+16], the quality assurance remains an open and challenging field of research [ER11; dMdCM+11; dCMC+14; LKL12].

Quality assurance, e.g., achieved via testing, is a crucial activity for the successful development of (complex) software systems [Har00; SLS11; AO16]. Therefore, testing is an integral part of software engineering process models such as the well-known V-model and, in general, consumes up to 50% of the available development resources [Har00; SLS11]. Due to the increasing software complexity as well as the limited testing resources, the application of testing for single-software systems is already challenging. However, for SPLs, software testing gets even more challenging as the variability represented by features has to be incorporated in the testing process resulting in another dimension of complexity [McG10; ER11; dMdCM+11; dCMC+14; LKL12]. For testing an SPL, all realizable variants have to be tested, where, in the worst case, the number of variants grows exponentially in the number of features. Furthermore, the inherent commonality shared between variants leads to redundant testing processes as reusable test cases are executed more than once to validate the same functionality again for different variants. Hence, the variability as well as the increased potential of testing redundancy impede the practical application of techniques for single-software testing, where each variant would be tested individually without taking the explicit knowledge about the shared commonality and variability into account [T'TK04; MI07; McG10; OMR10; ER11].

Existing SPL testing techniques follow different strategies to incorporate variability into the test process in order to facilitate efficient SPL testing by reducing the overall testing effort [T'TK04; McG10; OWE+11; ER11; dMdCM+11; dCMC+14; LKL12]. For instance, sample-based testing reduces the number of variants to be tested by selecting a representative subset of variants based on the adaptation of the concepts of combinatorial interaction testing [JHF12; AKT+16a; VAT+18]. The test of the representative subset allows for drawing conclusions for the quality of the complete SPL. In contrast, prioritization-based testing determines an optimized order of variants to be tested w.r.t. a

certain testing property such as feature coverage [ATM+14; SSR14; EBA+11; LJC+14; PSS+16; HPP+14]. An optimized testing order facilitates a resource-effective testing process as the process can stop after any variant of the given testing order such that the most important variants w.r.t. the test property under consideration are tested. Besides sample- and prioritization-based testing, family-based testing allows for the efficient creation of reusable test artifacts such as test cases [RKP+05; WSS08; Loc13; COL+11; BLB+15; Oli08; DPL+14]. Hence, test cases are derived solely once based on the reasoning about their reusability and not for every variant under test completely anew. However, independent from the application of those testing strategies each variant is still tested individually without exploiting the commonality and obtained test artifacts as well as test results between variants such that the testing redundancy is still a major drawback.

To overcome this drawback, the concepts of regression testing [YH12] are adapted to facilitate incremental SPL testing [TTK04; McG10; ER11; LLL+14; dMdCC+10; UGK+08; LLL+15; VBM15; DFG+17]. By exploiting the commonality shared between subsequently tested variants, regression-based SPL testing focuses on the differences between variants during their testing processes facilitating retest test selection [dMdCC+10; LLL+14; DSL+13], test-case prioritization [LLL+15], or the incremental creation of test cases [UGK+08; VBM15; DFG+17]. Especially, retest test selection techniques [dMdCC+10; LLL+14; DSL+13] tackle the testing redundancy by reusing test cases and test results during incremental SPL testing. Test cases are selected for their reexecution to revalidate that the differences between subsequent variants under test do not erroneously influence already tested functionality [Eng10b]. Hence, common functionality shared by subsequently tested variants and not influenced by the differences does not have to be retested such that the overall test effort can be strongly decreased according to McGregor [MI07; McG10]. However, those techniques [dMdCC+10; LLL+14; DSL+13] either perform a manual selection of reusable test cases or rather do not incorporate automated change impact analysis to support the selection process. Therefore, automated change-impact-based test-case selection for incremental SPL testing is still an open issue to facilitate the reduction of the testing redundancy by means of redundant test-case executions.

Besides the increasing complexity, software development has to face another aspect of the software life-cycle, namely software evolution [Leh96; SB99; GG08; MSC14]. Software evolution arising based on, e.g., changing requirements, is an inevitable process and cannot be neglected [SV02; GG08; MSC14]. Obviously, quality assurance has also to be performed after an SPL evolves [RE12b; RE12a]. Thus, SPL testing has to take the evolution of reusable development artifacts and their interdependencies into account resulting in an even more challenging activity of SPL engineering as not a single system is influenced by changes of an evolution step, but rather the complete set of variants. Existing SPL testing techniques mainly focus on the complexity dimension introduced by the variability of an SPL and abstract from evolution [TTK04; OWE+11; ER11; dMdCM+11; LKL12; dCMC+14]. Hence, the quality assurance of evolving SPLs achieved via testing is an open problem to be addressed, where according to Engström and Runeson [Eng10b; RE12a; RE12b], regression testing should be applied. However, there is no technique so far that applies regression testing for incremental testing of variants and versions of variants in order to facilitate efficient testing of evolving SPLs.

1.1 Contributions

In this thesis, we tackle the challenges of testing redundancy and evolution. Therefore, the main research question to be answered by our contributions is defined as follows:

How can we efficiently test evolving software product lines based on the reduction of redundant test-case executions?

For answering the main research question, we propose a testing framework that combines model-based testing [UL06; UPL12] as well as retest test selection as regression testing strategy [YH12] in order to exploit the reuse potential of test artifacts and test results for incremental testing of evolving SPLs. The combination of the two testing concepts is reasonable as the application of model-based and regression testing is both well-suited for SPL testing [Oli08; Loc13; Eng10b; McG10]. On the one hand, model-based testing facilitates the automated generation of test cases [UL06; UPL12] reusable between consecutively tested variants and versions of variants. On the other hand, retest test selection is a well-known strategy in regression testing [YH12; Eng10b] to reduce the set of test cases to be reexecuted in order to validate that already tested behavior which is shared between variants and versions of variants is not erroneously influenced by their differences. Furthermore, the combination of both testing concepts allows for the application of regression testing on the functional testing level such that systems can be tested in a regression-based manner in a black-box test setting [LW89; YH12], i.e., the source code of the system is not available for testing purposes.

For the combination of model-based and regression testing, three crucial activities which can also be seen as cornerstones of our framework have to be defined taking the dimensions of variability and evolution into account. As shown in Fig. 1.1, those activities are as follows:

Test Modeling builds the foundation for a successful application of model-based testing [UL06; UPL12; LPK+14]. A test model represents the abstract behavioral specification of a system under test and allows for the automated generation of test cases. In the context of evolving SPLs, a test-modeling formalism has to cope with variability and evolution in an integrated way to facilitate test case derivation as well as the reasoning about test case reusability between variants and versions of variants. The incorporation of both dimensions for integrated test modeling should also facilitate the application of analyses, e.g., to support retest test selection via change impact analysis. However, existing techniques for model-based SPL testing [Loc13; COL+11; LLL+14; DPL+14; VBM15; LLL+15; DFG+17; OWE+11] solely capture the variability of an SPL with their test-modeling formalisms. Furthermore, modeling techniques for handling SPL evolution handle variability and version information in different ways [ST00; LSK+13; NBA+15], are defined for source code [ALR+05; AMC+07], or do not facilitate the application of change impact analysis [ALR+05; AMC+05; HRR+12; SSA13a; KLL+14]. Hence, integrated test modeling for variants and versions of variants is an open issue to be addressed by our model-based regression testing framework.

Change Impact Analysis is essential for successful regression testing to guide retest test selection and test-case prioritization strategies [YH12]. Impact analysis identifies the influences of changes to already tested functionality of a software system. Such functionality influenced by changes indicates retest potentials to be retested during regression testing. In the context of evolving SPLs, change impact analysis is applicable in two scenarios for guiding retest test selection. First, by interpreting differences between variants and versions of variants as changes, we are able to identify their impact to already tested behavior indicating retest potentials to be retested during incremental regression-based SPL testing. Second, when stepping to the next SPL version under test, we are interested in the information, whether a variant to be tested gets modified or stays unchanged in order to guide the retest of complete variant

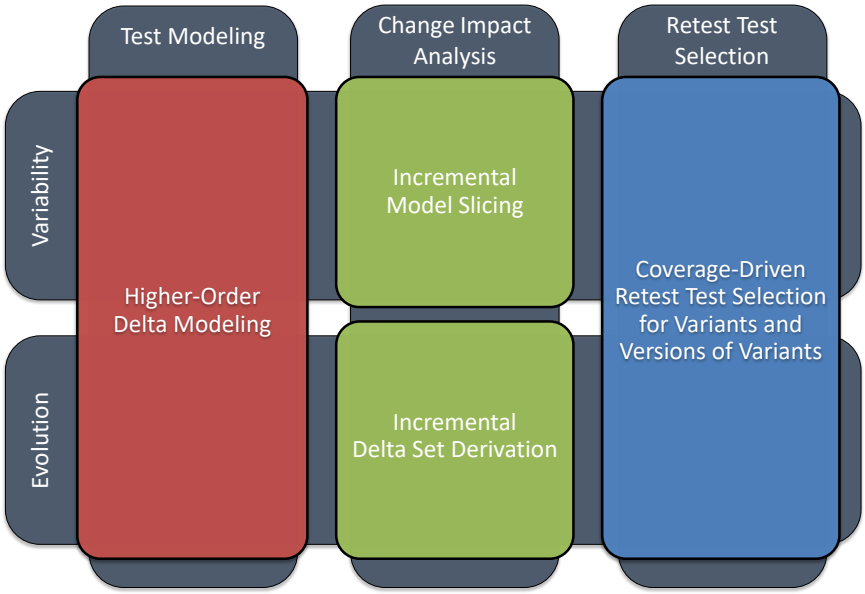


Figure 1.1: Overview of Contributions for Model-Based Software Product Line Regression Testing

versions. However, existing change impact analysis techniques applied for single-software regression testing are mainly code-based and also cannot handle variability [Boh96; Arn96; Leh11b; Leh11a; Bin98; YH12]. In addition, techniques providing change impact analysis for SPL evolution either detect changes to the variant set on the feature level or identify solely that a variant is modified, but do not determine how the original version and the modified version of the variant differ [NSS16; SBT16; TBK09; BKL+16]. Hence, change impact analysis facilitating (1) the detection of the impact of test-model changes to already tested behavior between subsequently tested variants and version of variants, and (2) the determination of the impact of an evolution step to the set of variants on the test-model level in terms of new, removed, unchanged, or modified variants is an open issue to be addressed by our model-based regression testing framework.

Retest Test Selection focuses on the identification of a subset of test cases to facilitate a retest of already tested functionality which is potentially affected erroneously by applied changes to a software system under test and their side-effects [YH12; KJM+17]. In the context of evolving SPLs, the adoption of retest test selection allows for tackling the potential redundancy during SPL testing introduced by the shared commonality by selecting reusable test cases for their reexecution [MI07; Eng10b]. For the determination whether a reusable test case has to be retested, the differences between subsequently tested variants or versions of variants and their impact to shared, yet already tested functionality has to be taken into account, e.g., based on the application of change impact analysis. However, existing SPL retest test selection techniques [dMdCC+10; LLL+14; DSL+13] either perform a manual selection of reusable test cases or do not incorporate automated change impact analysis to support the selection process. Hence, automated retest test selection guided by change impact analyses and applied to facilitate incremental testing of variants and versions of variants is an open issue to be addressed by our model-based regression testing framework.

For the realization of the framework, we contribute the following solutions for the described issues as depicted in Fig. 1.1. For test modeling, we propose *higher-order delta modeling* which is an extension of the transformational variability implementation technique delta modeling [CHS15; Sch10] and instantiate it for state machines to capture the variable behavioral specification, i.e., state machine test models of variants and versions of variants by the same means. Hence, a variable test model of an SPL version is represented by a delta model, where the differences between variant-specific test models are explicitly captured as transformations encapsulated in deltas, e.g., the addition and removal of states or transitions. For the incorporation of SPL evolution, we transform version-specific delta models by altering their encapsulated delta set via additions, removals, and modifications of deltas specified via higher-order deltas. Therefore, a higher-order delta model captures the complete evolution history of the behavioral specification of an SPL, where version-specific delta test models are derivable for incremental SPL testing. The application of higher-order delta modeling as test-modeling formalism is beneficial in two ways. First, we provide an integrated modeling formalism that incorporates variability as well as version information as first-class entities. Second, the explicit knowledge about the commonality and difference between subsequently tested variants and versions of variants by means of deltas facilitate change impact analysis as well as retest test selection as both concepts exploit the commonality and focus on the differences during their application.

For change impact analysis, we propose two techniques to be integrated in our model-based regression testing framework. First, we introduce *incremental model slicing* and its application as change impact analysis for guiding retest test selection during incremental testing of variants and versions of variants. The slicing technique exploits the specification of the shared commonality and the differences between subsequently tested variants and versions of variants by means of deltas to automatically identify changed execution dependencies during their incremental testing. Changed execution dependencies point to shared behavior potentially influenced by differences between the variant-specific state machine test models, e.g., the addition of a transition to a test model may influence the execution of an already existing transition such that the existing transition shows a different behavior when executed. Therefore, those changed dependencies refer to potential behavior to be retested during incremental testing of variants and versions of variants. Second, we present an *incremental delta set derivation* facilitating the reasoning about the application of higher-order deltas and its impact to the set of variants of an SPL version in terms of new, unchanged, or modified variants. Higher-order deltas specify how the delta set of a version-specific delta test model changes in terms of additions, removals, and modifications of deltas. The incremental delta set derivation exploits those changes to infer and reason about the respective changes on variant-specific delta sets. The reasoning process results in the categorization of how the variant set alters between consecutively tested SPL versions which is utilized in our framework to guide the incremental test process for evolving SPLs, e.g., modified variants are tested based on their previous version and unchanged variants are skipped as they are already tested.

For the application of retest test selection, we propose a retest coverage criterion to allow for *automated coverage-driven retest test selection* based on the results of the application of incremental model slicing as change impact analysis between consecutively tested variants as well as versions of variants. Therefore, we derive retest test goals by taking the identified changed execution dependencies into account. To ensure the coverage of the derived retest test goals, we automatically

select reusable test cases for their reexecution in order to validate that already tested behavior is not erroneously influenced when stepping to the next variant or version of a variant to be tested. Based on the coverage-driven selection of test cases, we reduce the number of test cases to be executed for testing evolving SPLs and, hence, tackle the testing redundancy by means of redundant test-case executions.

In the end, our framework represents the first technique that applies regression testing for efficient testing of individual SPL versions and subsequent SPL versions in an incremental way. Therefore, the framework unites the delta-oriented test-modeling formalism, the delta-oriented change impact analyses, and the coverage-driven retest test selection. During the incremental test process, the framework exploits the reuse potential of test artifacts and test results of already tested variants and versions of variants to reduce the overall testing effort by tackling the potential redundancy during testing of evolving SPLs. The resulting reduction of the overall test effort enables a more targeted use of the limited test resources and, hence, facilitates efficient quality assurance of variants and versions of variants achieved via model-based regression testing. We prototypically implement our framework as well as its three essential activities. Furthermore, we evaluate our contributions by means of three evolving SPLs to validate the applicability, effectiveness, and efficiency in controlled experiments.

1.2 Outline

The remainder of this thesis is structured as follows. In **Chapt. 2**, we describe the relevant background by means of an introduction in software testing as well as the development of software product lines. According to the three contributions proposed in this thesis, we also present their definitions divided in three respective chapters. Therefore, in **Chapt. 3**, we introduce the test-modeling formalism applied in the model-based regression testing framework to capture the behavioral specification of variants and versions of variants. In this context, the adaptation of delta modeling for state machines is defined as well as the extension to higher-order delta modeling facilitating the incorporation of SPL evolution. In **Chapt. 4**, we describe the two delta-oriented change impact analysis techniques required to guide the retest test selection process. As first analysis technique, we define incremental model slicing and its application for impact analysis between subsequently tested variants and versions of variants. The second analysis technique reasons about the application of higher-order deltas and its impact on the set of variants based on an incremental delta set derivation. In **Chapt. 5**, we introduce the automated coverage-driven retest test selection and its integration into the testing framework. In addition, the incremental workflow for model-based regression testing of subsequent SPL versions is presented. In the last **Chapt. 6**, we conclude the thesis and provide a discussion regarding potential future work. Additional material can be found within the appendix at the end of this thesis. We discuss related work as well as present the prototypical tool support and the evaluation w.r.t. the contributions in the respective chapters.

2 Background

In this chapter, we introduce the background for the contributions proposed in this thesis. First, we introduce the basic notions of general software testing and, in particular, for model-based as well as regression testing. Second, we describe software product lines and their development based on the process of software product line engineering and the application of variability management. In addition, we provide an overview on software product line testing.

2.1 Software Testing

Software testing is the most applied quality assurance technique for the successful development of software and consumes up to 50% of the available development resources [Har00; SLS11; AO16]. According to Bourque and Fairley [BF14]:

> "Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior." [BF14]

This definition comprises four important aspects [UL06]. First, the behavior of a software system under test (SUT) is verified against its specification based on the execution of test cases on the SUT which is defined as dynamic testing [SLS11]. In this context, a *test case* represents a certain execution scenario of the SUT based on defined *controllable inputs* and expected *observable outputs*. Besides dynamic software testing, static testing, static analysis, or formal verification can be applied for quality assurance, i.e., to validate and verify a SUT [SLS11; AO16]. Second, the expected outcome of a test-case execution by means of observable outputs of the SUT has to be specified also known as *test oracle problem* [BF14; BHM+15]. The definition of a test oracle is mainly performed manually by a test engineer based on the incorporation of suitable artifacts such as requirements and design models, but dependent on the provided development artifacts as well as the applied testing technique, the expected outcome for a test case can also be automatically derived [BHM+15]. Besides the test oracle, the observability of the behavior of a SUT w.r.t. the executions of test cases has to be ensured to allow for *test verdicts* in terms of *pass* and *fail* [BF14; SLS11]. Third, the result of the testing process differs w.r.t. a selected testing technique [BF14; SLS11; AO16]. Depending on a given set of test requirements and quality attributes, the testing technique defines the focus of the test process and, hence, specifies how to derive test cases for the validation of the set of test requirements and quality attributes. However, for all applicable testing techniques, the overall goal is to detect failures in the SUT. Fourth, testing can only be performed based on a finite set of test cases [BF14; SLS11; AO16]. As the domain of input values and their combinations result in a potentially infinite number of executions to be validated via test-case executions, the exhaustive testing of an SUT is practically infeasible. Therefore, *test end criteria* are to be used to facilitate the reasoning about the testing adequacy incorporating the limited testing resources as well as the fulfillment of given test requirements and quality attributes.

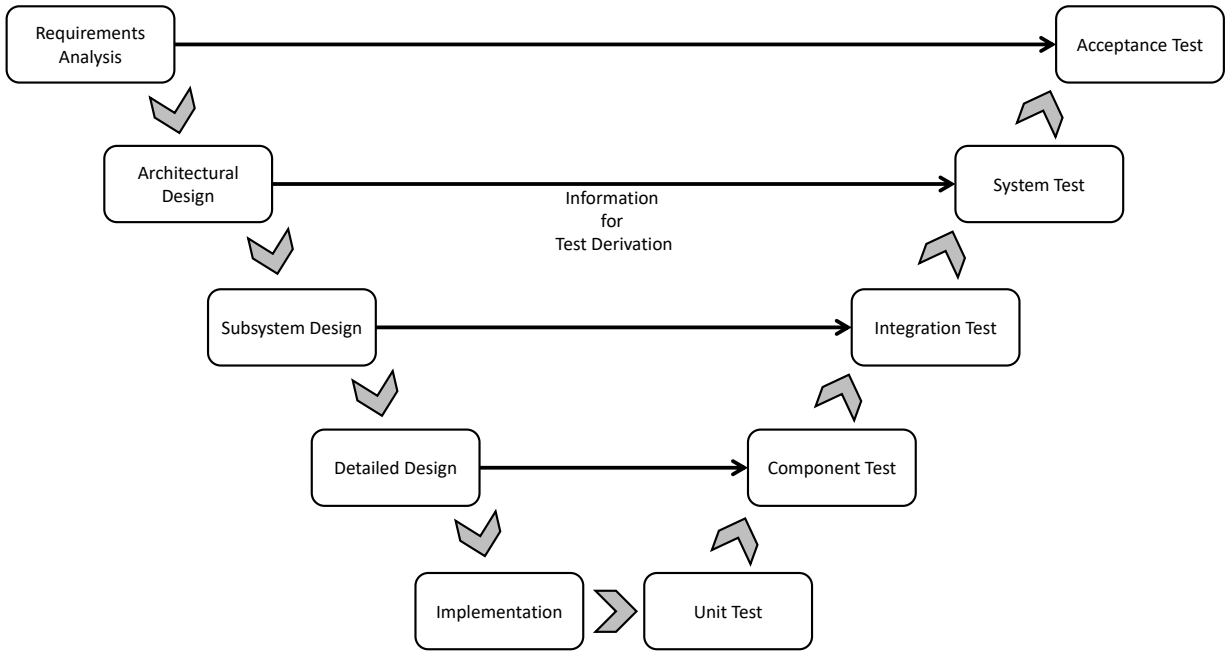


Figure 2.1: V-Model Software Engineering Process (according to Ammann and Offutt [AO16])

Testing Levels. Testing is a crucial activity and has to be applied on different levels w.r.t. the phases of the development process as early as possible [SLS11]. For instance, in Fig. 2.1 the V-Model is shown which is a commonly applied software engineering process model, where the branch on the right defines distinct levels for testing [Som10; AO16]. Those levels are related to development phases captured in the left branch of the V-Model such that respective software artifacts can be exploited to derive test requirements as well as test cases. The testing levels are as follows [AO16; SLS11]:

Unit Test Level. Unit testing is applied on the lowest level of the development phases, where it focuses on the validation of implemented software units such as methods of a class against their specifications.

Component Test Level. This level focuses on the testing of software components against their specifications, where development artifacts of the detailed design phase are taken into account. A software component is represented, e.g., by a class or a group of classes implemented to provide a specific functionality of the complete SUT.

Integration Test Level. Integration testing is concerned with the validation of the correct interaction of components based on the specification defined in the subsystem design phase. During the testing process, components are incrementally integrated in the subsystem to be tested in order to facilitate the detection of faults in the component interfaces and, hence, in the communication between components.

System Test Level. This test level focuses on the testing of the complete software system against the customer requirements. Hence, the testing process has to answer whether all requirements are suitably satisfied by the SUT.

Acceptance Test Level. Acceptance testing is performed in the presence of the customer to build trust in the quality of the SUT by validating it against the requirements and application scenarios of the customer defined in a respective contract.

In general, the distinct testing levels of the V-Model are also adapted or similarly defined for other software engineering process models or agile techniques [SLS11; Sch04]. In this thesis, we propose a model-based SPL regression testing framework as introduced in Chapt. 5 which is applied on the component testing level such that component variants as well as their versions are tested against their behavioral specifications.

Black-Box vs. White-Box Testing. Dynamic testing techniques applied on the different testing levels such as the component testing level are categorized in the classes of *black-box* and *white-box testing* [SLS11]. This distinction is made based on the information and artifacts which are incorporated or rather available for the test process of an SUT. For white-box testing, also called structure- or code-based testing, the internals of a test object such as the source code of a component are accessible for the derivation of test cases [SLS11]. Therefore, test adequacy criteria in terms of statement or branch coverage are applicable to guide the test-case derivation process. The resulting set of test cases ensures that the complete source code is executed at least once such that, in addition to the detection of failures, the detection of dead, i.e., non-executable code is facilitated. In contrast, for black-box testing, also called functional- or specification-based testing, the internals of a test object are not incorporated, e.g, as they are potentially not accessible. Hence, the derivation of test cases has to be performed solely based on the set of requirements and also other design artifacts exploitable as specification of, e.g., a software component to be tested [SLS11]. As no information about the internals of a SUT is exploitable to define test adequacy criteria, the specified input domain of the SUT is, in general, used to define coverage criteria, e.g., by deriving equivalence classes of the input domain w.r.t. the expected outputs to be covered by respective test cases.

In this thesis, we apply model-based component testing which belongs to the class of black-box test techniques and combine it with regression testing, i.e., retest test selection, to specify a framework for efficient testing of evolving SPLs. The framework exploits the commonality shared between subsequently tested variants and focuses on their differences to reason about the reexecution of reusable test cases such that a reduction of redundant test-case executions is achieved. In the following subsections, we describe model-based testing and regression testing in more detail.

2.1.1 Model-Based Testing

Model-based testing is a specification-based testing technique, where executable test cases are derived based on a test model [UPL12]. Utting and Legeard [UL06] summarized model-based testing as follows:

> "Model-based testing is the automation of the design of black-box tests." [UL06]

A *test model* represents the abstract behavioral specification of an SUT created based on the incorporation of the requirements of an SUT. Different modeling formalisms are applied for test modeling in the literature [DSV+07] such as finite state machines, activity diagrams, sequence diagrams etc. Those formalisms have in common that they provide a formal semantics and also facilitate the modeling of relevant characteristics of an SUT in an abstract form [PSM12]. Furthermore, the testing level as well as the defined test requirements influence the selection of the formalism [DSV+07]. In general, a test model captures the behavior of the SUT by means of the specification of the reaction on *controllable inputs* with *expected outputs* [UL06; UPL12]. Hence, an SUT conforms to its specification if it reacts to inputs with the respective expected outputs.

To validate the conformance, test cases are automatically derived from the test model [UL06; UPL12; DSV+07]. A *test case* represents a path through the test model such that it defines the sequence of inputs in combination with the expected reactions of the system. Hence, a derived test case comprises the test oracle information required for an automated analysis of its execution to provide a test verdict [UL06; UPL12]. Depending on the abstraction level of a test model, the derived test cases may be too abstract and have to be concretized to allow for their execution on the SUT.

Similar to other test derivation techniques, test adequacy criteria are used to guide the derivation process. For model-based testing, mainly structural coverage criteria are applied [UL06; UPL12], e.g., *all-states* or *all-transitions* if the behavioral specification is defined by a state machine test model. Based on a selected coverage criterion, a set of *test goals* is derived and exploited to generate test cases, where each test goal, e.g., a transition, has to be traversed via the test model path of a generated test case in order to be covered. For each test goal, at least one test case must be generated to be collected in a *test suite*.

The test model created based on the requirements of an SUT, the set of test goals derived via the application of a structural coverage criterion, and the test suite generated based on the test model as well as the test goal set are summarized as *model-based test artifacts* used for testing an SUT. In this thesis, we apply state machines as test-modeling formalism (cf. Chapt. 3) as common for model-based testing [UL06; LPK+14] and use all-transition coverage to guide the test-case creation (cf. Chapt. 5). For a general overview on model-based testing and automated test-case generation based on state machine test models, we refer to the literature [UPL12; DSV+07; AS12; SK13; ABC+13].

2.1.2 Regression Testing

Regression testing is a technique applied after an SUT has been changed due to maintenance, fault correction, or extension of the provided functionality in order to validate that changes are implemented correctly and do not erroneously affect already tested functionality [AO16; SLS11]. A precise definition is given by IEEE (IEEE-Std-610.12-1990 1990) [90]:

> "Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirement." [90]

Regression testing is independent from a certain testing level and is further applicable in black-box as well as white-box test scenarios [SLS11; LW89]. For regression testing to be effective, the information about changes of an SUT has to be provided [LW89]. Otherwise, the SUT is interpreted as completely modified or as a new software system such that a retest of the SUT cannot be performed. In this context, Leung and White [LW89] defined two types of regression testing, namely *corrective regression testing* and *progressive regression testing*. In corrective regression testing, the specification between the original SUT and its modified version remains unchanged which is the case, e.g., in a fault correction scenario. In contrast, for progressive regression testing, the specification along with the SUT gets changed, e.g., due to the extension of the provided functionality of the SUT. However, both types facilitate the retest of an SUT, but differ in the amount of retest potentials limited to the parts of the SUT which are common between its two versions [LW89].

A complete retest of an SUT based on the execution of all test cases that are valid for the modified version of the SUT is called *retest-all* [AO16; SLS11; YH12]. In general, the application of retest-

all is not practicable due to the limited resources of the testing process of an SUT [LW89; YH12]. Hence, existing regression testing techniques focus on different testing aspects to reduce the regression testing effort and can be categorized in the following three classes of regression testing strategies [YH12]:

Test-Suite Minimization aims at the reduction of the number of test cases to be reexecuted for a retest of the modified SUT by identifying and omitting redundant test cases. A *redundant test case* is defined as a test case which provides the same coverage of, e.g., statements or transitions as other test cases contained in the same test suite [LW89]. Hence, the execution of redundant test cases solely consumes testing resources, but does not reveal further failures as those test cases which provide the same coverage such that redundant test cases can be removed from the test suite to exploit the limited testing resources more efficiently.

Test-Case Prioritization is concerned with the problem to find an optimal order for the execution of test cases to facilitate the maximization of test properties such as the coverage or the early fault detection. Prioritization strategies are cost-effective as the testing process of an SUT can stop at some arbitrary point in time and prioritization strategies ensure the execution of the most important test cases w.r.t. the test property under consideration.

Test-Case Selection also aims at the reduction of the number of test cases to be reexecuted for a retest of the modified SUT. In contrast to test-suite minimization techniques, test-case selection focuses on the identification of a subset of test cases of a given test suite to facilitate a retest of already tested functionality which is potentially affected erroneously by applied changes to the SUT and their side-effects.

In this thesis, we focus on the third strategy, i.e., retest test selection, to allow for progressive regression testing of evolving SPLs. We exploit the commonality shared between subsequently tested variants as well as versions of variants and focus on their differences to reason about the retest of test cases in order to reduce the number of redundantly executed test cases in variant-specific testing processes. We introduce our retest test selection applied for model-based SPL regression testing in Chapt. 5 and describe the *retest test selection problem* [RH96; YH12] in the following.

Retest Test Selection Problem. Rothermel and Harrold [RH96] provided a definition for the retest test selection problem:

Given: Program P and its modified Version P' as well as a test suite TS for P executable on P'
Find: Subset TS' of TS for testing P'

For the determination of the subset of test cases to be reexecuted for retesting an SUT, the change information and the impact of changes to already tested functionality has to be taken into account [LW89; YH12; SLS11]. Thus, the application of change impact analysis is crucial to identify those parts of an SUT which are already tested, but affected by changes. Those parts have to be retested based on the selection of respective modification-traversing test cases in order to validate that no erroneous side-effects are introduced [RH96; YH12]. Furthermore, the selection allows for a classification of the test suite used for the test of the original SUT in order to (re)test the modified SUT version by means of four categories [LW89; YH12]:

Obsolete The category of obsolete test cases captures all test cases which were executable on the original SUT, but are not executable on the modified SUT. Those test cases are removed from the test suite used for the retest of the modified SUT.

Reusable All test cases executable on the original SUT and the modified SUT version are categorized as reusable. The set of reusable test cases builds the basis for retest test selection.

Retestable The category of retestable test cases comprises all modification-traversing test cases that are selected from the set of reusable test cases based on the application of change impact analyses.

New The changes applied on the original SUT to obtain its modified version also introduce new functionality that cannot be tested based on the set of reusable test cases. Hence, new test cases are required for testing new functionality and to provide the respective coverage of, e.g., source code or test model.

At the end of the selection and categorization process, new test cases are executed in addition to the retestable test cases for retesting the modified version of an SUT based on its original version. For a general overview on regression testing, we refer to the literature [LW89; RH96; ERS10; YH12; KJM+17].

2.2 Software Product Lines

Due the increasing prevalence of software-intensive systems and further the request for individualized software systems, the development of single-software systems reaches its limits [CN01; PBvdL05; LSR07]. This has led to a shift in the development trend for software systems. Instead of developing each customer-specific system from scratch, software development follows the strategy of mass customization and, therefore, facilitates the mass production of customer-individualized software systems based on the large-scale reuse of software artifacts in the development process [PBvdL05; LSR07]. A paradigm to accomplish mass customization is the development of *software product lines* (SPL). An SPL represents a family of similar software systems also called *variants*, where the commonality and variability between variants is explicitly documented [PBvdL05]. Another definition for SPLs is given by Clements and Northrop [CN01]:

> "A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [CN01]

The definition comprises two important aspects. First, a set of *features* $F = \{f_0, \ldots, f_n\}$ is used to specify the commonality and variability and, therefore, the similarity between variants $v \in \mathbb{V}$ of an SPL [CN01; PBvdL05; LSR07]. By $\mathbb{V} = \{v_0, \ldots, v_m\}$, we refer to the set of variants that can be created from an SPL. Varying definitions for the term feature exist as documented by Apel et al. [ABK+16], where the definition is dependent on the abstraction level and phase of the development process. In this thesis, we apply the definition given by Kang et al. [KCH+90], where a *feature* represents a customer-visible functionality of an SPL. Second, the development of variants by exploiting and, hence, reusing an existing set of *core assets* [CN01; PBvdL05; LSR07]. Core assets represent reusable software artifacts of all phases of the software development life-cycle such as design models, software components, test cases etc. The set of core assets, also known as *platform*, builds the basis for the large-scale reuse of artifacts during the creation of customer-individualized software variants.

The concept of SPLs is not new and was already successfully applied in industry, where among others the mobile phone devision of Nokia and the medical systems devision of Philips are examples [LSR07; Wei08; ABK+16]. The successful realization of SPLs provides various benefits [PBvdL05].

Based on the development of the platform, software artifacts are implemented once as their reuse is planned and specified for an SPL resulting in a reduction of the development costs. In addition, the large-scale reuse facilitated by the platform reduces the time for the realization of variants and, therefore, reduces the time-to-market. Another main benefit is given by an increased quality. Based on the reuse of software artifacts for creating distinct variants of an SPL, the artifacts are tested thoroughly in different environments as well as in different combinations with other artifacts. As a consequence, the confidence and, thus, the quality of variants and even of the complete SPL gets increased. Furthermore, the correction of a failure in a reusable artifact which was detected during the test of a certain variant improves the quality of all variants that are build by the respective artifact. This scenario also indicates a reduction in the maintenance effort as failures have to be corrected once and not for all affected variants separately. However, those benefits are not achieved automatically, i.e., a large initial investment is required to identify the commonality and variability by means of features and to build the platform of reusable software artifacts [PBvdL05; LSR07].

2.2.1 Software Product Line Engineering

For the successful development of SPLs, Pohl et al. [PBvdL05] proposed the software product line engineering (SPLE) process which is further divided in the subprocesses of *domain engineering* and *application engineering* as shown in Fig. 2.2. According to van der Linden et al. [LSR07], the division in subprocesses facilitates the exploitation of the reuse potential specified by the platform in two ways:

> "Software product line engineering relies on the fundamental distinction of development for reuse and development with reuse." [LSR07]

Therefore, the domain engineering is responsible for the *development for reuse*, whereas the application engineering allows for the *development with reuse* [PBvdL05; LSR07]. We describe both subprocesses in the following paragraphs. The SPLE process is, in general, independent from a concrete adoption strategy [Kru02] such that an SPL can be developed (1) completely from scratch by following a *proactive* strategy, (2) by incrementally extending an already existing SPL based on a *reactive* strategy, or (3) based on the incorporation of an existing set of software variants following an *extractive* strategy.

Domain Engineering. Pohl et al. [PBvdL05] define the role of the domain engineering process as follows:

> "Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realised." [PBvdL05]

Therefore, this SPLE subprocess is responsible for the identification of features as well as the creation of the reusable platform. To achieve this, the domain engineering is divided into five phases which are related to respective phases for the development of single-software systems [PBvdL05]:

Product Management identifies and documents the domain, i.e., market segment, an SPL is developed for. Based on the overall business goals used as input, the commonality and variability in terms of features is defined. For the feature definition, the process further incorporates existing variants, their requirements, and their development artifacts. As the result, a product road map of the SPL is defined capturing the identified features and also those development

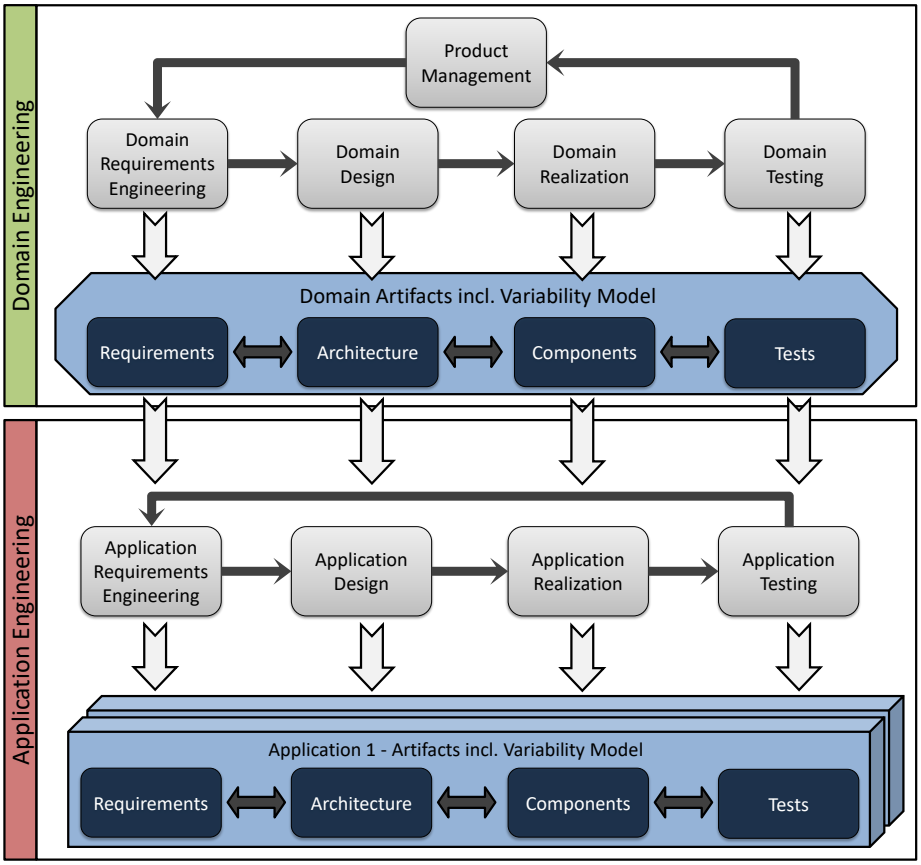


Figure 2.2: Software Product Line Engineering Process (according to Pohl et al. [PBvdL05])

artifacts which are used in the domain design to initialize the reference architecture. The road map also facilitates the future planning of the SPL development based on a defined release schedule of features or variants.

Domain Requirements Engineering gathers and documents the common and variable requirements of all variants of the SPL to be developed based on the product road map. The common and variable features captured in the road map are refined to derive a detailed set of respective requirements. The set of domain requirements builds the basis for the subsequent domain engineering processes, i.e., the domain design, realization, and testing. As output of the phase, a variability model is defined specifying the detailed commonality and variability by means of features and requirements and further documenting the variation points of the SPL.

Domain Design creates the reference architecture specifying the common as well as the variable software functionality and structure from a technical point of view. For the creation of the architecture, the domain design incorporates the variability model, the domain requirements, and the domain artifacts already identified in the product management subprocess. Domain requirements are directly linked to software parts to ensure traceability. Furthermore, the set of reusable domain artifacts is identified. The reference architecture represents the foundation to facilitate mass customization. In addition to the reference architecture, this subprocess provides a refined variability model such that a categorization in terms of internal and external variability is given.

Domain Realization implements reusable software artifacts by means of components and interfaces for the common as well as the variable software parts based on the reference architecture. In addition, configuration mechanisms are introduced to facilitate the generation of variants by selecting reusable software artifacts in the application realization subprocess.

Domain Testing validates and verifies the results of the domain engineering subprocesses, in general, but mainly the reusable software artifacts implemented in the domain realisation process against the specification of the SPL to find faults in an early development phase. The specification is given by the reference architecture, the variability model, and the domain requirements. Furthermore, the domain testing process derives reusable test artifacts during the validation of reusable software artifacts in order to reduce the effort for testing generated variants in the application testing process.

Application Engineering. According to Pohl et al. [PBvdL05], the role of the application engineering is defined as follows:

> "Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability." [PBvdL05]

Similar to domain engineering, the application engineering process is divided into phases such that for each of the domain engineering phases except the product management, a respective counterpart is defined for the application engineering process [PBvdL05]:

Application Requirements Engineering takes the demands of a customer into account to gather and document the requirements of the variant to be created. Therefore, the customer's requirements are examined to identify and reuse as much of the domain requirements for the creation. In case the domain requirements are not sufficient to realize the variant, customer-specific requirements are derived and added to the set of reused domain requirements. By comparing the customer-specific requirements and the domain artifacts, the differences are identifiable and exploited to reason about the feasibility of the creation of the variant. Furthermore, the domain variability model is refined to obtain a variant-specific variability model based on the binding of variation points. Both the variant-specific variability model and requirements build the basis for the subsequent processes of application engineering to create the variant a customer demands.

Application Design produces a variant-specific architecture based on the reference architecture as well as the refined variability model and customer requirements. Therefore, variation points of the reference architecture are bound to select and reuse respective software parts that are required for the creation of a variant. In case the reference architecture is not sufficient to fulfill the customer's requirements, the resulting variant-specific architecture is further adapted to ensure the fulfillment. Based on the reusability of software parts given by the reference architecture, an SPL developer can focus on the required variant-specific adaptations such that the development time can get reduced.

Application Realization creates the variant a customer demands by selecting and assembling reusable development artifacts created in the domain realization process. In addition, a configuration of the selected artifacts may be required to bind the internal variability. Similar to the application design process, customer-specific software artifacts are potentially imple-

mented to fulfill the respective requirements. The resulting variant is complete and, therefore, executable, but has to be tested in order to be delivered to the customer. Besides the executable variant, the application realization provides a detailed design of the created variant for documentation purposes.

Application Testing validates and verifies the created customer-specific variant against its specification. The test process is applied on different testing levels, e.g., the unit or integration testing level, and further aims at the fulfillment of the quality adequacy criteria defined by the testing level under consideration. Again, the reusability of domain artifacts is exploited by selecting reusable test artifacts derived in the domain testing process. For the validation of customer-specific software parts, respective customer-specific test artifacts are additionally created. As output of the application testing process, the executed test cases, their test results as well as identified defects are documented in test reports. The test documentation facilitates the reasoning about the achieved quality of the created customer-specific variant. Furthermore, the detection of defects improves the overall quality of the SPL under development as the correction of the defect located in a reusable artifact is fixed for all variants.

Both SPLE subprocesses are iteratively executed, i.e., new or changing requirements can be incorporated in the domain engineering and distinct customer-specific variants can be created in the application engineering. The domain and application engineering processes are further intertwined. For instance, customer-specific adaptations or implementations of development artifacts can be integrated in the set of reusable domain artifacts by reapplying the domain engineering process after a variant is created based on the demands of a customer in the application engineering process.

In this thesis, we focus on the domain and application testing phases as we propose a framework for model-based SPL regression testing. In the domain testing phase, we define reusable test models, i.e., the behavioral specifications of variants under test, using the variability implementation technique delta modeling as described in Chapt. 3. Based on the application of model-based testing, we automatically derive reusable test cases from the test models. For application testing, we exploit the concepts of regression testing such that (1) we incorporate the reusable delta-oriented test models for the application of change impact analysis as defined in Chapt. 4 and (2) we perform retest test selection guided by the results of the change impact analysis in order to reduce the redundancy in test-case executions during the testing process of an SPL as introduced in Chapt. 5.

2.2.2 Variability Management in the Problem and Solution Space

In the SPLE process, the identification of commonality and variability by means of features, the implementation of the platform as well as the creation of customer-individualized software based on the reuse of software artifacts selected from the platform are the main activities which are realized based on the application of *variability management* [SJ04; SD07; CAA09]. According to Schmid and John [SJ04], variability management is defined as follows:

> "Variability management encompasses the activities of explicitly representing variability in software artifacts throughout the lifecycle, managing dependences among different variabilities, and supporting the instantiation of the variabilities." [SJ04]

For a successful variability management, the knowledge about commonality and variability has to be documented as well as implemented based on its integration or its specification in software

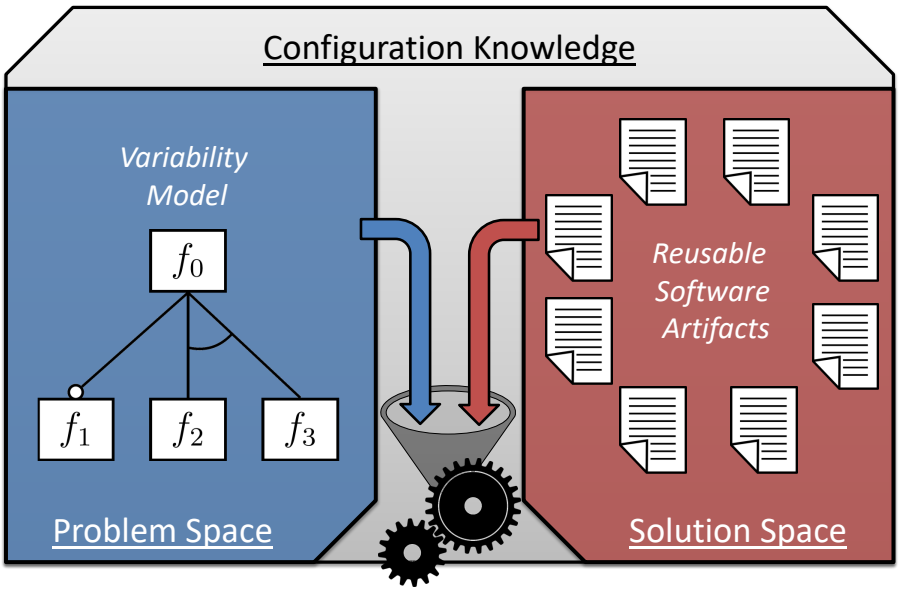


Figure 2.3: Relation of Variability Modeling in the Problem and Solution Space

artifacts which is also known as *variability modeling* [SD07; SRC+12]. As shown in Fig. 2.3, the process of variability modeling is subdivided w.r.t. the problem as well as solution space of an SPL [CE00]. In the *problem space*, the common and variable configuration options and their relations are defined in a variability model resulting in the specification of the set of variants $\mathbb{V}$ of an SPL by means of the valid configuration space [CE00]. Hence, variability modeling in the problem space is responsible for capturing the commonality and variability based on respective artifacts, e.g., feature models [KCH+90; BSR10] to facilitate the configuration, i.e., the binding of variability for the creation of a certain variant [SRC+12; CE00]. In the *solution space*, the software artifacts and their reusability are defined based on the explicit knowledge about the commonality and variability of an SPL [CE00]. Therefore, variability modeling in the solution space also named variability realization mechanism is responsible for the implementation of the common and variable software artifacts to be reused for the creation of a certain variant [SRC+12; CE00]. Furthermore, the problem and solution space and, therefore, their respective artifacts are linked to each other in terms of a mapping defined by the *configuration knowledge*. The configuration knowledge is gathered during the process of variability management and comprises the information which reusable software artifacts have to be selected from the solution space based on a given configuration from the problem space and how the selected artifacts are assembled for the creation of a certain variant [SRC+12; CE00].

In the following sections, we describe techniques for variability modeling in both spaces and discuss how this thesis is related to variability modeling. For a general overview on variability management as well as variability modeling, we refer to the literature [CAA09; SD07; SRC+12].

Variability Modeling in the Problem Space

The problem space defines the commonality and variability of an SPL by means of features used as configuration options [CE00]. For the specification and documentation of the knowledge of the commonality and variability in a variability model, different techniques were proposed in the literature [CAA09; SD07; SRC+12], where feature modeling [KCH+90; SHT06; BSR10; MTS+17] is

mainly applied. Besides feature modeling, the orthogonal variability model [PBvdL05], the common variability language [HMO+08], and decision modeling [SRG11] are established approaches to be used for variability modeling in the problem space.

In this thesis, we apply feature modeling [KCH+90; SHT06; BSR10; MTS+17] to capture the commonality and variability of an SPL to be tested as well as to facilitate the definition of delta-oriented test models to be reusable during the testing process of an SPL. Hence, we describe the concept of feature modeling in the following.

Feature Modeling. The notion of feature models (FM) was introduced by Kang et al. [KCH+90] in the context of *feature-oriented domain analysis*. A *feature model fm* captures all features $F = \{f_0, \ldots, f_n\}$ identified for an SPL and their interrelations. Furthermore, a feature model specifies the allowed combinations of features to facilitate the definition of the set of variants $\mathbb{V}$ of an SPL. A valid combination of features represents a feature configuration $F_v \subseteq F$, i.e., a subset of features selected from the feature model, and is mapped to a respective variant $v \in \mathbb{V}$. Therefore, each variant $v \in \mathbb{V}$ is explicitly defined based on its feature configuration $F_v \in F_{\mathbb{V}}$ of all derivable feature configurations of an SPL. For a formal definition of feature models by means of an abstract syntax, semantics, and potential extensions, we refer to the literature [SHT06; BSR10; Loc13].

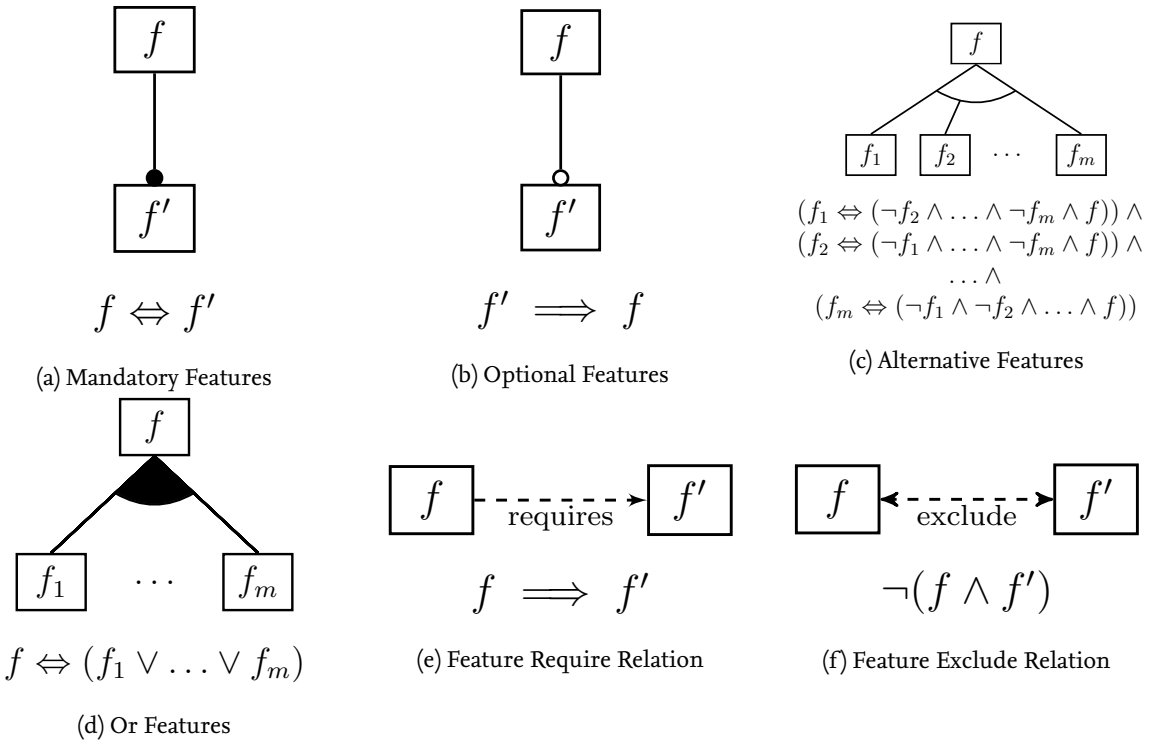
Feature models are often represented graphically as a feature diagram [SHT06], whereas the representation as a propositional formula [Man02; Bat05; BSR10] is also common. A *feature diagram* captures the features of an SPL in a tree-like structure, where they are related to each other based on typed parent-child relations. In Fig. 2.4, the graphical representations of the common types of parent-child relations used in feature diagrams are shown, where further the propositional formulas for a respective typed relation are also provided below the graphical representation. The types of parent-child relations are as follows [KCH+90; CE00; SHT06; BSR10]:

Mandatory Features. The functionality represented by a mandatory feature f' has to be selected for a feature configuration F_v if its parent f is also selected. As shown in Fig. 2.4a, a mandatory feature is indicated by a black circle on top of its feature (node). The formula representation is defined by using the biconditional logic operator $\Leftrightarrow$, i.e., the parent feature f implies the selection of the child feature f' and vice versa.

Optional Features. For an optional feature f', its selection is not obligatory if the parent feature f is selected for a feature configuration F_v. An optional feature is represented by a white circle on top of its feature (node) as depicted in Fig. 2.4b. The parent-child relation for optional features is defined as an implication such that the optional child feature f' implies the selection of the parent feature f.

Alternative Features. A group of alternative features f_1 to f_m indicates a specialization of the respective parent feature f. If the parent feature is selected for a feature configuration F_v, exactly one alternative feature f_i from the alternative group has to be selected for F_v. As shown in Fig. 2.4c, an alternative group is represented by a white semicircle connecting all alternative features f_1 to f_m of the group. The formula representation is given by the conjunction of alternative-feature-specific formulas such that an alternative feature f_i is selected if and only if the parent feature f is selected and none of the other alternative features $\neg f_j$ contained in the alternative group.

Or Features. A group of or features f_1 to f_m indicates optional increments of the functionality of the respective parent feature f. If the parent feature is selected for a feature configuration F_v,



(a) Mandatory Features (b) Optional Features (c) Alternative Features

(d) Or Features (e) Feature Require Relation (f) Feature Exclude Relation

Figure 2.4: Overview on the Typed Parent-Child Relation of a Feature Model and the Respective Representation as Propositional Formula

at least one feature f_i from the or group has to be selected for F_v. As shown in Fig. 2.4d, an or group is represented by a black semicircle connecting all or features f_1 to f_m of the group. The formula representation is defined by using the biconditional logic operator $\Leftrightarrow$, i.e., the parent feature f implies the selection of at least one child feature f_i denoted by the disjunction of the or features and vice versa.

Feature models can contain abstract features such as the special abstract root feature. Abstract features are used in a feature model to facilitate the tree-like structure, but have no mapping to reusable software artifacts [TKE+11]. Furthermore, a feature model comprises also *cross-tree constraints*, i.e., requires and exclude constraints, or rather more general constraints expressible by propositional formulas [CE00]. In general, constraints are used in a feature model to restrict or predefine the potential combination of features. As shown in Fig. 2.4e, a *requires constraint* is represented by a dashed one-directional arrow connecting two features f and f' such that the selection of feature f implies the selection of feature f' for the same feature configuration F_v. In contrast, an *exclude constraint* depicted in Fig. 2.4f is denoted by a dashed bidirectional arrow between two features f and f', where the selection of feature f for a feature configuration F_v implies the non-selection of f', i.e., both features cannot be selected for the same feature configuration.

In the variability management process, feature models represent variability models in the problem space (cf. Fig. 2.3) to capture the variability and commonality and, therefore, the possible configuration options. Based on a feature configuration F_v and the configuration knowledge, a respective variant $v \in \mathbb{V}$ is created by selecting and assembling reusable software artifacts from the platform.

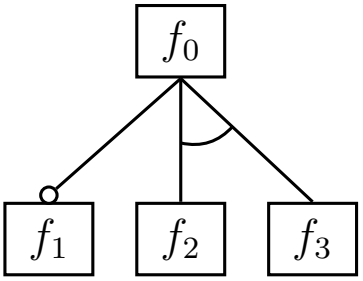


Figure 2.5: Sample Feature Model

Example 2.1: Feature Modeling

Consider the sample feature model *fm* in Fig. 2.5 comprising the root feature f_0, the optional feature f_1, and the alternative features f_2 as well as f_3. From this feature model, four feature configurations $F_{\mathbb{V}} = \{F_{v_0}, \ldots, F_{v_3}\}$ are derivable and, therefore, four variants $\mathbb{V} = \{v_0, \ldots, v_3\}$ can be created by the sample SPL which is used in this thesis as running example. The four feature configurations are defined by

$F_{v_0} = \{f_0, f_2\}$,
$F_{v_1} = \{f_0, f_2, f_1\}$,
$F_{v_2} = \{f_0, f_3\}$, and
$F_{v_3} = \{f_0, f_3, f_1\}$.

In addition to the representation as feature diagram, the sample feature model *fm* can be translated to a propostional formula such as

$$[\![fm]\!]_{\mathbb{B}} = f_0 \wedge (f_1 \implies f_0) \wedge (f_2 \Leftrightarrow (\neg f_3 \wedge f_0)) \wedge (f_3 \Leftrightarrow (\neg f_2 \wedge f_0)),$$

where $[\![.]\!]_{\mathbb{B}}$ represents the function to translate a feature model in its propositional formula representation as defined in the literature [Man02; Bat05; BSR10].

Variability Modeling in the Solution Space

For the implementation of reusable artifacts, various variability implementation techniques exist in the literature also known as variability realization mechanisms [SD07; SRC+12]. Besides the implementation, those techniques also facilitate the specification how reusable artifacts are assembled for the creation of a variant. For the specification, a given feature configuration as well as the mapping between features and software artifacts captured in the configuration knowledge are incorporated as shown in Fig. 2.3. In general, the set of existing techniques is categorized in the classes of annotative, compositional, and transformational variability implementation techniques [SRC+12].

Annotative Implementation Techniques. Annotative techniques focus on a superimposed artifact representation often called 150% representation, where all variant-specific artifacts of the same type of an SPL are merged together [CA05; Gom06; SRC+12]. To distinguish between parts of the superimposed artifact and to reason about their reusability for certain variants, annotations, e.g., feature expressions or stereotypes are used. For the creation of a variant, the annotations comprised in the superimposed representation are evaluated based on a respective feature configuration such that those parts are removed, where the annotation is not

satisfied by the given feature configuration. Hence, annotative techniques implement negative variability as all parts of the superimposed representation are removed which are not valid for a certain variant. As result, the variant-specific artifact such as the source code or a design model is obtained [SRC+12].

Compositional Implementation Techniques. For compositional techniques, reusable fragments of artifacts also called modules are implemented and linked to features [SRC+12; ABK+16]. A variant-specific artifact is created by selecting and composing all fragments that are linked to features contained in the variant-specific feature configuration. Another strategy defined for compositional implementation techniques is the refinement of a base module via reusable fragments, where the base module captures the common functionality shared between variants [BSR04; SRC+12]. Therefore, compositional techniques provide a realization of positive variability as a variant-specific artifact is incrementally created by composing selected artifact fragments or by refining a given base module [BSR04; SRC+12; ABK+16].

Transformational Implementation Techniques. Similar to compositional techniques, transformational approaches require a base module also called core module to define transformation rules by means of additions, removals, and modifications of reusable artifacts in order to transform the core into a variant-specific artifact [SRC+12; HMO+08; CHS15]. As a special case, the core module can be empty such that the variant-specific artifact is mainly created based on additions [SD10]. To specify for which variant a transformation rule has to be applied to the core, each rule is mapped to a feature or a feature expression used as application condition. Based on a given feature configuration of a certain variant to be created, the application conditions of all defined transformation rules are evaluated such that a rule is applied to the core if its condition is satisfied by the configuration. Transformational implementation techniques are in a certain way an extension of compositional techniques as additions to the core represent refinements, but the specification of modifications as well as removals are also possible to transform the core into a variant-specific artifact [SRC+12; SD10].

Variability implementation techniques are instantiable for different types of development artifacts such as source code, design models etc. and, hence, are not bound to a certain type facilitating a general application of variability modeling in the solution space [SD07; SRC+12]. In this thesis, we adapt delta modeling [CHS15; Sch10], which belongs to the class of transformational approaches, to define reusable state machine test models for variants and versions of variants. Delta modeling [CHS15; Sch10] allows for the explicit specification of differences between variants and versions of variants and is already applied for test modeling in the context of incremental SPL testing [LSK+12; LLS+12; LLL+14; LLL+15; LMT+16; LLA+16; VBM15; DFG+17; LNT+19]. The delta-oriented test models build the basis for the combination of model-based as well as regression testing in a framework for efficient testing of evolving SPLs. We introduce the adaptation of delta modeling as test-modeling formalism in Chapt. 3 and further describe our testing framework in Chapt. 5.

2.2.3 Software Product Line Testing

The application of testing for single-software systems is already a challenging task [Har00; SLS11; AO16]. However, for SPLs, testing gets even more challenging as variability has to be incorporated in the testing process [McG10; ER11; dMdCM+11; dCMC+14; LKL12]. Engström and Runeson deduced three major challenges for SPL testing based on their survey of existing SPL techniques [ER11]:

1. **Large Number of Variants and Tests**: The complete test of an SPL requires the test of every derivable variant. This is hard to achieve or even infeasible as the number of variants typically grows exponentially in the number of features. Furthermore, for the complete test, a large number of test cases is required. Due to the inherent commonality shared between variants, the reusability of test cases also leads to redundant testing processes as the same test cases reusable for different variants are executed to validate the same functionality again.
2. **Reusable Components and Concrete Variants**: The second challenge is concerned with the trade-off regarding the division of the effort to be spent for domain testing and for application testing. Reusable domain artifacts such as reusable components have to be tested to ensure their correct (variable) functionality. However, its integration and interaction with other reusable artifacts in a concrete variant under test is also important. Especially, the interaction with other reusable artifacts created for certain features of the SPL is another challenge for SPL testing also known as *feature interaction problem* [Zav93]. A feature interaction arises if the functionality of a feature is influenced by the presence or absence of another feature. As feature interactions are not always intended, SPL testing has to cope with erroneously introduced feature interactions which can be detected in most cases during the test of a concrete variant and rather not by an isolated test of a reusable domain artifacts.
3. **Variability**: SPL testing has to focus on the variability by means of features, their interrelations, and its binding times. As described for the second challenge, the implementation of the functionality of a feature has to be validated in domain testing and their interrelations by means of feature interactions in the application testing phase. Testing also has to validate that variability is bound correctly, e.g., the functionality of a feature which is not selected for a variant has to be absent in this variant.

By applying single-software testing techniques, each variant would be tested individually without taking the explicit knowledge about the shared commonality and variability into account also known as product-by-product strategy [TTK04] or contra-SPL-philosophy [OMR10]. The variability as well as the increased potential of testing redundancy impede the practical application of techniques for single-software testing [McG10; ER11]. Therefore, techniques for SPL testing have to take the explicit knowledge about commonality and variability into account to be efficiently applicable and have to exploit the paradigm of large-scale reuse also for the reuse of test artifacts [MI07; McG10; ER11; LKL12]. Existing SPL testing techniques follow different strategies to tackle the described challenges as well as to incorporate the commonality and variability for their application [TTK04; McG10; OWE+11; ER11; dMdCM+11; dCMC+14; LKL12]. The existing techniques can be categorized according to their testing strategies in the classes of sample-based, prioritization-based, regression-based, and family-based testing and are defined as follows:

Sample-Based Product-Line Testing aims at the reduction of the overall SPL testing effort by selecting a representative subset of variants from the complete variant space of an SPL under test [LFR+15; LFC+16; VAT+18]. To guide the selection process, mainly criteria adapted from combinatorial interaction testing are applied such as pairwise feature interaction coverage [OMR10; AKT+16a] or, in general, t-wise feature coverage [PSK+10; JHF12]. Thus, variants are selected for the representative subset which provide the largest increase in t-wise coverage until the complete coverage is ensured. For an overview on sample-based testing, we refer to respective surveys [LFR+15; LFC+16; VAT+18].

Prioritization-Based Product-Line Testing focuses on the maximization of a certain testing property such as the (t-wise) feature coverage or the early fault detection rate by identifying an optimized testing order of variants under test [ATM+14; SSR14; EBA+11; LJC+14; PSS+16; HPP+14]. For the maximization, the dissimilarity of variants in terms of, e.g., their feature configurations [ATM+14; HPP+14; DPL+16] are examined such that the next variant under test is the most dissimilar to all already integrated variants in the testing order. Prioritization techniques are resource-effective as the SPL testing process can stop after any variant of the given testing order such that the most important variants w.r.t. the test property under consideration are tested. We provide a more detailed discussion about SPL prioritization techniques in Sect. 5.5.

Regression-Based Product-Line Testing aims at the reduction of the overall SPL testing effort by focusing on the reusability of test artifacts and by tackling the redundant execution of test cases [dMdCC+10; UGK+08; BL14; VBM15; DFG+17; LLL+14; LLL+15; DSL+13]. Therefore, respective techniques adapt the concept of regression testing strategies for incremental SPL testing. The commonality between subsequently tested variants is exploited in order to focus on the differences between tested variants during their testing processes facilitating test-case selection [dMdCC+10; LLL+14; DSL+13; BL14], test-case prioritization [LLL+15], or the incremental creation of test suits [UGK+08; VBM15; DFG+17]. We provide a more detailed discussion about regression-based techniques in Sect. 5.5.

Family-Based Product-Line Testing exploits the application of annotative implementation techniques for the realization of test and development artifacts to allow for the creation of reusable test artifacts such as test models and test cases [RKP+05; WSS08; Loc13; COL+11; Oli08; DPL+14] as well as the variational execution of test cases [MWK+16] in order to reduce the overall SPL testing effort. Family-based testing is mainly applied in the context of model-based SPL testing, where an annotative test model, i.e., 150% test model, is used for automated test-case generation [WSS08; COL+11; LBL+14]. In contrast to the generation of test cases for every variant under test anew, the 150% test model allows for the reasoning about the reusability of test cases for distinct variants during their derivation by incorporating the annotations comprised in the test model.

The testing techniques of those categories are not solely applicable individually, but rather can be combined to facilitate efficient SPL testing. Besides those categories, there exist SPL testing techniques that adapt the concepts of mutation testing [JH11; Off11] to improve the testing effectiveness [HPP+13; LS14; DPC+14; AGV15; ABT+16; RBR+15; RLB+18]. Therefore, mutants are derived based on the application of variability-aware mutation operators representing faulty versions of variants. Mutation-based SPL testing exploits the derivation of such mutants (1) to measure the effectiveness of, e.g., sampling strategies [HPP+13; RLB+18], or (2) to perform fault-based generation of test configurations, i.e., certain variants to be tested [AGV15; RBR+15]. For a general overview on SPL testing, we refer to respective surveys [TTK04; McG10; OWE+11; ER11; dMdCM+11; dCMC+14; LKL12].

In this thesis, we focus on the first major challenge which was proposed by Engström and Runeson [ER11] and also on the evolution of SPLs. Therefore, we propose a model-based SPL regression testing framework that exploits the commonality shared between subsequently tested variants as well as versions of variants and focuses on their differences to reason about the reexecution of

reusable test cases such that a reduction of redundant test-case executions is achieved. Based on the application of regression testing by means of retest test selection for incremental testing of variants and versions of variants, our framework belongs to the category of regression-based SPL testing techniques.

3 Delta-Oriented Test Modeling for Variants and Versions of Variants

The content of this chapter shares material with work published in [LLS+12], [LKS16], [LMT+16], [NLS18], and [LNT+19].

Contribution

We adapt the existing variability implementation technique delta modeling to capture the variability of variants and versions of variants by the same means. We extend delta modeling by lifting its concept to transform version-specific delta models by altering the encapsulated delta set via additions, removals, and modifications of deltas. The extension provides benefits and limitations for product-line evolution and, thus, for the application as test-modeling formalism facilitating the efficient testing of subsequent software product-line versions under test. We discuss those benefits as well as limitations and show the applicability of our extension based on three evolving model-based software product lines.

In this chapter, we introduce the test-modeling formalism to be incorporated in our model-based regression testing framework (cf. Chapt. 5). Test modeling builds the foundation for a successful application of model-based testing techniques [UL06; UPL12; LPK+14]. A test model represents the behavioral specification of a system under test and allows for the automated generation of test cases [UL06; UPL12; LPK+14]. In the context of SPLs, for every variant a respective test model is required. Due to the vast number of potential variants, the individual definition of variant-specific test models is infeasible. Furthermore, the shared commonality between variants results in redundant test-modeling steps.

Hence, variability implementation techniques [SRC+12] applied for the development of reusable domain artifacts during SPLE [PBvdL05] can be adopted for test-modeling purposes. For instance, in the context of model-based SPL testing, annotative or transformational approaches [SRC+12] are mainly applied for test modeling [Loc13; COL+11; LSK+12; LLS+12; LLL+14; DPL+14; DPL+15; VBM15; BLB+15; LMT+16; DFG+17; LNT+19; LKL12; OWE+11; ER11], i.e., to specify the variable behavior of all variants of an SPL. Annotative techniques, such as 150% modeling [SRC+12; CA05], further facilitate the application of family-based analyses [TAK+14], e.g., to allow for efficient SPL test-suite generation [COL+11; BLB+15]. In contrast, transformational approaches, such as delta modeling [Sch10; CHS15], allow for the explicit specification of differences between variants enabling incremental regression-based testing [LSK+12; LLS+12; LLL+14; LLL+15; LMT+16; LLA+16; VBM15; DFG+17; LNT+19] with automated change impact analysis [LMT+16; LNT+19].

However, product lines evolve over time [SB99; BP14; MSC14], e.g., due to their continued development or changing requirements. The evolution may impact all development artifacts, their interdependencies, and their variant-specific composition due to the respective changes. To cor-

respond to the new SPL version, the variable test model, i.e., the behavioral specification of the evolving SPL, must also evolve. Existing techniques for managing SPL evolution in the solution space [ST00; ALR+05; AMC+07; DGR+10; SSA13a; HRR+12; LSK+13; KLL+14; NBA+15] (cf. Sect. 3.5) have at least one of the following four limitations in order to be applied as test-modeling formalism for evolving SPLs:

(1) They handle variability and version information in different ways [ST00; LSK+13; NBA+15] impeding the readability. – In addition to the dimension introduced by variability, evolution introduces a second dimension increasing the complexity of a test model. By coping with both dimensions by the same means and, therefore, handling variant and version information as first-class entities, the comprehensibility is improved.

(2) They are artifact-specific, e.g., solely applicable for source code [ALR+05; AMC+07]. – The adaptability of a modeling technique for different test-model types facilitates its application in different testing phases, e.g., for component and integration testing. Furthermore, in a model-based regression testing framework (cf. Chapt. 5), code-based techniques are not applicable.

(3) They do not capture the complete evolution history, but tackle evolution steps individually without taking the history into account [HRR+12; KLL+14; NBA+15]. – The documentation of the evolution history is important for the traceability of changes made by evolution steps. In addition, the information about the history can be taken into account for testing purposes.

(4) They do not facilitate analysis w.r.t. SPL evolution [ALR+05; AMC+05; HRR+12; SSA13a; KLL+14], e.g., change impact analysis. – Based on the inherent complexity of a test model representing the behavioral specification of an evolving SPL, the test process has to be supported by analyses. In the context of evolution, change impact analysis facilitates (1) the detection and classification of variants between subsequent SPL versions under test influenced by an evolution step in terms of new, modified, or unchanged variants (cf. Sect. 4.2), and (2) the detection of influenced dependencies based on changes between the original variant and its modified version (cf. Sect. 4.1) indicating behavior to be retested by our framework (cf. Chapt. 5).

Those four limitations to overcome are, therefore, crucial to allow for efficient testing of evolving product lines. Hence, SPL evolution has to be captured in a concise, expressive, and flexible way by means of an integrated (test-)modeling formalism (1) for handling both, variability and evolution in the same way, (2) to be adaptable and, thus, applicable for various (test-model) artifact types, (3) for documenting the complete evolution history, and (4) for facilitating the automated analysis about the evolution impact.

We propose *higher-order delta modeling* an extension of delta modeling [CHS15; Sch10] as integrated modeling formalism to address the four requirements and discuss the benefits and limitations for its application in SPLE [PBvdL05], in general, and for supporting the test process of evolving SPLs, in particular. Delta modeling [CHS15; Sch10], already adapted for different types of domain artifacts [LLL+14; SSA13b; DSL+13; HKM+13; KHS+14; LMB+14; CDD+16] and used for test modeling in the context of efficient SPL testing [LSK+12; LLS+12; LLL+14; LLL+15; LMT+16; VBM15; DFG+17; LNT+19], is well-suited to capture not only the variability, but also the version information of an evolving product line as first-class entities. Based on higher-order deltas encapsulating evolution operations, i.e., additions/removals/modifications of deltas, we evolve a delta model representing the variable test model for one SPL version in time to correspond to the delta model of the subsequent SPL version. The evolution history of a delta-oriented SPL is documented by means of

higher-order delta models. By analyzing the application of higher-order deltas (cf. Chapt. 4), we are able to reason about the evolution impact in terms of new, unchanged, or modified variants which is exploited to facilitate efficient regression testing of evolving SPLs (cf. Chapt. 5).

The remainder of this chapter is structured as follows. First, we describe state machine modeling in Sect. 3.1 building the foundation for the introduction of test modeling for variants and versions of variants. Second, we explain delta modeling as existing variability modeling technique and its instantiation for state machines in Sect. 3.2. Third, we propose our extension, i.e., higher-order delta modeling, capturing the solution space variability of evolving SPLs in Sect. 3.3. Fourth, we describe our tool support for higher-order delta modeling and show the applicability of our extension based on three evolving model-based SPLs in Sect. 3.4. Fifth, we discuss related work on SPL evolution in Sect. 3.5. Finally, we conclude the chapter in Sect. 3.6.

3.1 State Machine Test Modeling

In this section, we describe the modeling formalism used as foundation to instantiate delta modeling [CHS15; Sch10] and its extension to manage SPL evolution (cf. Sect. 3.3) of model-based SPLs. As our product-line regression testing framework (cf. Chapt. 5) combines model-based testing [UL06; UPL12; LPK+14] and retest test selection as regression testing strategy [YH12], we use *state machines* as base type for domain artifacts and apply it as test-modeling formalism. State machines are a well-established modeling formalism already employed in the context of single and variant-rich software systems for model-driven development [Har87; HP98; Obj09; Col06] as well as quality assurance, e.g., model-based testing [UL06; UPL12; Wei10; LPK+14; Loc13; LTW+14; SV08]. A state machine specifies the input-output behavior of a system, i.e., external input events are used to *control* the system by stimulating the behavior such that the system reacts with *observable* output events. Furthermore, the state machine dialect used in this thesis incorporates the decomposition of behavior based on the concepts of hierarchy and concurrency [Har87] and abstracts from variables for communication purposes. The abstraction from variables is reasonable in two ways. On the one hand, a state machine applied for test modeling represents the abstract system behavior and, therefore, does not have to specify the behavior on the same level of granularity as, e.g., required for design models [Obj09] used to generate source code [GHP02; PD07]. On the other hand, we are able to encode the read-/write-access of variables via corresponding events such that also complex behavior is specifiable solely based on events [Mil89]. In the following paragraphs, we describe the abstract syntax as well as the execution semantics of the state machine dialect which we apply as test-modeling formalism. The abstract syntax is based on the definitions by Wang et al. [JWZ02] and Lochau [Loc13], whereas the execution semantics is based on the work of Harel and Naamad [HN96] and Lochau [Loc13].

3.1.1 Abstract Syntax for State Machines

To facilitate the modeling of complex systems incorporating hierarchy and concurrency [Har87], we define a *state machine* as a composition of state machine regions. A *state machine region* specifies the behavior of a part of a software system by capturing a set of corresponding abstract computation states as well as a set of transitions representing the potential transfer between those states. A transition is triggered by an event, e.g., an external input event, and potentially broadcasts newly generated events, e.g., as observable reaction, as output. Both, the triggering event and the event

broadcast define the label of a transition. Therefore, a region also comprises a set of events specifying the interface of a region in terms of input, internal, and output events.

Definition 3.1: State Machine Region

Let $\mathcal{E}$ be the universe of all events potentially used to specify the input-output behavior of software systems. Furthermore, $\mathcal{S}$ and $\mathcal{T} = \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ represent the universe of all possible abstract computation states and transitions, respectively, whereas $\mathcal{L} = \mathcal{E} \times \mathcal{P}(\mathcal{E})$ denotes the universe of transition labels defined over $\mathcal{E}$. A *state machine region* $r = (S, s_0, E, L, T)$ is a 5-tuple, where

- $S = \{s_0, \ldots, s_m\}, S \subset \mathcal{S}$ is a finite set of *states*,
- $s_0 \in S$ is the initial state,
- $E = (E_I \cup E_\tau \cup E_O) \subset \mathcal{E}$ is a finite set of *events* defined by disjoint sets of input events E_I, internal events E_τ, and output events E_O,
- $L \subseteq E \times \mathcal{P}(E), L \subset \mathcal{L}$ is a finite set of *transition labels*, and
- $T \subseteq S \times L \times S, T \subset \mathcal{T}$ is a labeled *transition relation*, where further holds that

$$\forall t = (s, e, \{\ldots\}, s') \in T : \neg\exists t' = (s, e', \{\ldots\}, s'') \in T, t \neq t' : e = e',$$

i.e., for each transition, there exist no other transition that starts in the same source state s and has the same triggering event e.

Input events *control* the system's behavior by stimulating, i.e., triggering transitions. Internal events are used to *control* internal behavior and to *synchronize* concurrent/hierarchical regions of the system by applying them as triggering event and in event broadcasts. Output events denote the *observable* reaction of a system to input stimuli. Based on the property of the transition relation, i.e., all transitions starting in the same source state have distinct triggering events, a region facilitates the specification of deterministic system behavior. For a more readable representation of transition labels, we write $(s, e/\{e', \ldots\}, s')$ or (s, l, s') for transitions $(s, e, \{e', \ldots\}, s')$ in the following.

State machine regions or simply regions capture the behavior of a specific part of the system. Hence, a state machine is composed of a set of regions to specify the complete behavior, where a designated *root region* exists representing the entry point of the system's behavior. To facilitate the definition of hierarchy as well as concurrency, we map states to their subregions as well as regions to their parent region via respective hierarchy functions. Based on the mapping and, therefore, the hierarchy functions, we ensure that, except for the root region representing the highest hierarchy level, (1) every region has one particular parent region, (2) every region is mapped to one particular state denoting a subregion relation, and (3) regions are not cyclically nested. In addition, a state machine comprises a set of events specified by the state machine regions allowing for the communication with the environment as well as internally between regions.

Definition 3.2: State Machine

Let $\mathcal{R}$ be the universe of all regions defined over $\mathcal{E}$. The set $S_R = \bigcup_{r \in R} S_r$ represents the union of region-specific sets S_r of states with $r \in R$ and $R \subset \mathcal{R}$ is the set of state machine regions. The set $E_r = (E_I^r \cup E_\tau^r \cup E_O^r)$ denotes a region-specific event set defined by its disjoint

input, internal, and output event sets. A *state machine* $sm = (R, r_0, \psi, \chi, E)$ is a 5-tuple, where

- $R = \{r_0, \ldots, r_m\}$ is a finite set of *regions*,
- $r_0 \in R$ is the *root region*,
- $\psi : S_R \rightarrow \mathcal{P}(R)$ is a *sub-hierarchy function* ensuring the following properties, where ψ^* represent the recursive determination of subregions of a state down to the lowest hierarchy level denoting the transitive closure of nested subregions:
  1. $\forall s \in S_R : \psi(s) = R_s, R_s \in \mathcal{P}(R)$, i.e., each state $s \in S_R$ is directly mapped to its potential subregions
  2. $\forall r \in R \setminus \{r_0\} : \exists s \in S_{r_0} : r \in \psi^*(s)$, i.e., except for the root region, every region is connected to the state machine based on a mapping to a state starting in a state s of the root region r_0
  3. $\forall r \in R \setminus \{r_0\} : \neg\exists s, s' \in S_{r_0}, s \neq s' : \psi^*(s) \cap \psi^*(s') \neq \emptyset$, i.e., except for the root region, every region is mapped to one particular state of the root region via the transitive closure of the sub-hierarchy.
- $\chi : R \setminus \{r_0\} \rightarrow R$ is a *parent-hierarchy function* with

$$\chi(r) = r' \Leftrightarrow \exists s \in S_R : r \in \psi(s) \wedge s \in S_{r'}$$

  such that the following properties are ensured, where χ^* represent the recursive determination of parent regions of a region up to the highest hierarchy level denoting the transitive closure of predecessor regions:
  1. $\forall r \in R \setminus \{r_0\} : \exists r' \in R \setminus \{r\} : \chi(r) = r'$, i.e., except for the root region, every region has one particular parent region
  2. $\forall r \in R : r \notin \chi^*(r)$, i.e., every region is not a predecessor region of itsself preventing from cyclic hierarchy relations
- $E = E_I \cup E_\tau \cup E_O = \bigcup_{r \in R} E_r$ is a finite set of *events* such that $E_I = \bigcup_{r \in R} E_I^r$, $E_\tau = \bigcup_{r \in R} E_\tau^r$, and $E_O = \bigcup_{r \in R} E_O^r$ holds.

The hierarchy functions ψ and χ facilitate the definition of a state machine as rooted tree. In addition, based on the definition of a region (cf. Def. 3.1), transitions always connect two states of the same region such that it is not possible to define hierarchy crossing transitions.

Example 3.1: State Machine Modeling

Consider the sample state machine $sm = (R, r_0, \psi, \chi, E)$ in Fig. 3.1. The state machine is defined by

- $R = \{\texttt{Root}, \texttt{A}, \texttt{B}, \texttt{C}\}$ with
  $\texttt{Root} = (S, s_0, T, E, L) = (\{s_0, s_1\}, s_0, \{e_3^\tau, e_8^O\}, \{(e_3^\tau / \{e_8^O\})\}, \{t_6\})$,
  $\texttt{A} = (\{a_1, a_2\}, a_1, \{e_1^I, e_2^\tau, e_3^\tau, e_4^O\}, \{(e_1^I / \{e_2^\tau, e_4^O\}), (e_2^\tau / \{e_3^\tau\})\}, \{t_1, t_2\})$,
  $\texttt{B} = (\{b_1, b_2, b_3\}, b_1, \{e_3^\tau, e_5^I, e_6^O, e_7^I\}, \{(e_5^I / \{e_6^O\}), (e_7^I / \{e_3^\tau\})\}, \{t_3, t_4, t_5\})$, and
  $\texttt{C} = (\{c_1, c_2\}, c_1, \{e_9^I, e_{10}^O, e_{11}^I, e_{12}^O\}, \{(e_9^I / \{e_{10}^O\}), (e_{11}^I / \{e_{12}^O\})\}, \{t_7, t_8\})$,
- $r_0 = \texttt{Root}$,
- $\psi : \psi(s_0) = \{\texttt{A}, \texttt{B}\}; \psi(s_1) = \{\texttt{C}\}$, and

- $\chi : \chi(\texttt{A}) = \{\texttt{Root}\}; \chi(\texttt{B}) = \{\texttt{Root}\}; \chi(\texttt{C}) = \{\texttt{Root}\}$, and
- $E = \{e_1^I, e_5^I, e_7^I, e_9^I, e_{11}^I\} \cup \{e_2^\tau, e_3^\tau\} \cup \{e_4^O, e_6^O, e_8^O, e_{10}^O, e_{12}^O\}$.

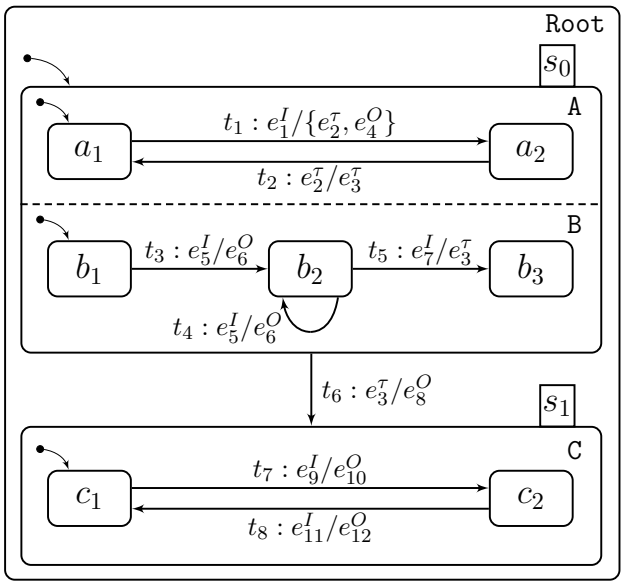


Figure 3.1: Sample State Machine *sm*

Based on the abstract syntax of our state machine dialect, we instantiate delta modeling [CHS15; Sch10] for state machines in Sect. 3.2 by incorporating the predefined entities, e.g., regions, for the delta definition. To define our change impact analysis (cf. Chapt. 4), especially the incremental model slicing, we also require the execution semantics of our modeling formalism. Depending on the applied semantics, the interpretation of dependencies we are reasoning about during the slicing process may differ. We describe the execution semantics used in this thesis in the following.

3.1.2 Execution Semantics for State Machines

The abstract syntax of our state machine dialect defines the entities, e.g., states and transitions, which are used to specify the abstract input-output behavior of a software system, but it does not define how the software system represented by the state machine reacts on occurring input events with respective output events. Therefore, we have to provide an execution semantics for our state machine dialect. The execution semantics defines the interpretation of the modeled behavior in terms of input-output reactions by specifying how events are consumed and, hence, how the transfer between states takes place. For an overview about existing state machine dialects and their execution semantics, we refer to the literature [vdBee94; CD05; Esh09; Loc13]. The execution semantics used in this thesis is based on the definitions of Harel and Naamad [HN96] as well as Lochau [Loc13].

During the execution, a state machine has at any point in time a specific configuration representing the abstract state in which the respective system is in. An abstract system state is thereby not represented by solely a single computational state $s \in S_R$ of a state machine, but rather represented by several regions and their contained states due to the decomposition by regions and depending on the present execution. Therefore, a *state machine configuration* encapsulates the active part of a state machine *sm* in terms of a set of active regions and their active states, where the root region

r_0 and an active state of r_0 is always part of a configuration as the root region represents the entry point of the modeled system behavior.

Definition 3.3: State Machine Configuration

Let $\mathcal{C}_{sm} \subset \mathcal{P}(R \times S_R)$ be the set of all state machine configurations for a given state machine *sm*. A *state machine configuration* $C = \{(r_0, s_{r_0}), \ldots, (r_j, s_{r_j})\} \in \mathcal{P}(R \times S_R)$ of a state machine $sm = (R, r_0, \psi, \chi, E)$ is a set of tuples of active regions $r \in R$ and their active states $s \in S_r$ such that the following holds:

- $\neg\exists C \in \mathcal{C}_{sm} : (r_0, s_{r_0}) \notin C$, i.e., the root region and one of its states is always part of a state machine configuration
- $\forall (r_j, s_{r_j}) \in C \setminus \{(r_0, s_{r_0})\} : \exists (r', s_{r'}) \in C : \chi(r_j) = r' \wedge r_j \in \psi(s_{r'})$, i.e., except for the root region, for every active region the respective parent region and the current active state are part of the configuration
- $\forall (r_j, s_{r_j}) \in C : \forall r' \in \psi(s_{r_j}) : (r', s_{r'}) \in C$, i.e., for every active region the respective subregions and their current active states are part of the configuration
- $\forall (r_j, s_{r_j}) \in C \setminus \{(r_0, s_{r_0})\} : r_j \in \psi^*(s_{r_0})$, i.e., except for the root region, every active region is part of the transitive closure of the subregion of the current active state of the root region

A state machine configuration $C = \{(r_0, s_{r_0}), \ldots, (r_j, s_{r_j})\} \in \mathcal{C}_{sm}$ is called *initial configuration* C_0 iff the following holds in addition:

- $\forall (r, s_r) \in C_0 : s_r = s_0^r$, i.e., for all active regions r, their active states $s_r \in S_r$ are the initial states $s_r = s_0^r$
- $\forall (r, s_r) \in C_0 \setminus \{(r_0, s_0^{r_0})\} : r \in \psi^*(s_0^{r_0})$, i.e., except for the root region r_0, every active region r is part of the transitive closure of subregions of the initial state $s_0^{r_0}$ of r_0

To define the reaction of a system to occurring events $e \in E_I \cup E_\tau$, we step from the current configuration $C \in \mathcal{C}_{sm}$ to the next one $C' \in \mathcal{C}_{sm}$ by consuming the events and providing the respective output. A *state machine step* captures the maximal set of conflict-free transitions which are executable in the current configuration based on given events. A transition is executable if its source state is active in the current configuration and if it is triggered by one of the occurring events. Similar to Harel and Naamad [HN96], we assume that an event can be consumed solely in the next step after its creation, e.g., based on the broadcast of an executed transition, resulting in a durability of one step. If the event cannot be consumed by a transition, the event is neglected and the state machine has to wait for another input event to step to the next configuration. Furthermore, two transitions are in conflict if they cannot both be executed in the same step. This is the case if both are triggered and (1) they have the same source state, or (2) they are contained in different hierarchy levels. Case (1) denotes a nondeterminism defined within a region which is excluded due to the definition of the transition relation of regions facilitating solely the specification of deterministic behavior (cf. Def. 3.1). However, Case (2) is possible and also wanted as the execution of a transition on a higher hierarchy level allows to interrupt the behavior of its subregions. To resolve the potential conflict, we incorporate the transition which is contained in the higher hierarchy level in the next step similar to the Statemate semantics of Harel and Naamad [HN96].

Definition 3.4: State Machine Step

Let $E^I_{st} \subset E_I \cup E_\tau$ be the set of input events of a state machine step that allows for the transfer between configurations either sent by the environment (E_I) or internally sent based on the execution of transitions in the prior step (E_τ). In contrast, $E^O_{st} \subset E_O \cup E_\tau$ represents the set of output events of a state machine step either sent to the environment (E_O) or internally sent for synchronization purposes (E_τ). By $l^{bc}_t = \{e', \ldots\}$, we refer to the broadcast set of events of a transition $t = (s, l_t, s')$ with transition label $l_t = (e, l^{bc}_t)$, i.e., all events that are generated and (internally) sent by executing a transition. A *state machine step* $st = (C_i, E^I_{st}, T_{st}, E^O_{st}, C_j)$ of a state machine $sm = (R, r_0, \psi, \chi, E)$ is defined, where the following holds

- $C_i, C_j \in \mathcal{C}_{sm}$,
- $T_{st} = \{t_1, \ldots, t_l\} \subseteq T_R$ such that

$$\begin{aligned} t \in T_{st} \Leftrightarrow & (\exists (r,s) \in C_i : t = (s, l, s')) \wedge (\exists e \in E^I_{st} : t = (s, e/\{e', \ldots\}, s')) \wedge \\ & (\neg \exists (r', s'') \in C_i, r \in \psi^*(s''), t \in T_r : \exists t' \in T_{r'} : t' = (s'', l, s''') \wedge \\ & \exists e \in E^I_{st} : t' = (s'', e/\{e', \ldots\}, s''')) \end{aligned}$$

- the output event set is generated by $E^O_{st} = \bigcup_{t \in T_{st}} l^{bc}_t$, and
- the configuration C_j is constructed via

$$\begin{aligned} C_j = & (C_i \setminus (\{(r,s) \in C_i | t = (s, l, s') \in T_{st} \cap T_r\} \cup \\ & \{(r,s) \in C_i | \exists t = (s', l, s'') \in T_{st} : r \in \psi^*(s')\})) \\ & \cup \{(r,s) \in \mathcal{C}_{sm} | t = (s', l, s) \in T_{st} \cap T_r\} \cup \\ & \{(r,s) \in \mathcal{C}_{sm} | \exists t = (s', l, s'') \in T_{st} : r \in \psi^*(s'') \wedge s = s^r_0\} \end{aligned}$$

We define the set of all possible steps for a state machine sm by $\mathcal{ST}_{sm}$.

Based on the configuration and step definitions, we are able to specify a run of a state machine. A *state machine run* is specified by a sequence of state machine steps and denotes a potential execution of the respective software system represented by a state machine.

Definition 3.5: State Machine Run

A *state machine run* $smr = (st_0, \ldots, st_l)$ is defined as sequence of state machine steps, where further holds that

- $st_0 = (C_0, E^I_{st_0}, T_{st_0}, E^O_{st_0}, C)$, i.e., the run starts in the initial configuration C_0,
- $\forall st_i = (C_{i-1}, E^I_{st_i}, T_{st_i}, E^O_{st_i}, C_i), 1 \leq i < l : st_{i+1} = (C_i, E^I_{st_{i+1}}, T_{st_{i+1}}, E^O_{st_{i+1}}, C_{i+1})$, i.e., the target configuration of the preceding step st_i is the source configuration of the successive step st_{i+1}, and
- $E^O_{st_i} \cap E^I_{st_{i+1}} \subset E_\tau$, i.e., besides the input events sent from the environment, the generated internal events of the last step st_i are used as inputs for the next step st_{i+1}.

We define the set of all state machine runs also representing the set of all potential system executions of a given state machine sm by $\mathcal{SMR}_{sm}$.

A state machine captures the complete abstract behavior of a software system, whereas a specific state machine run represents a potential system execution. Every state machine run further denotes a certain path within a state machine, e.g., used for the derivation of test cases [UL06; UPL12]. As state machines are composed of regions, we first define a path within a region and, upon this, the path of a state machine specified by a certain state machine run. A *region path* starts in the region-specific initial state and subsequent transitions are connected via source and target states to describe a path.

Definition 3.6: State Machine Region Path

Let T_r^* be the the set of all paths of a region r. A *region path* $\rho_r = (t_1, \ldots, t_k) \in T_r^*$ of length k of a region $r \in R$ of state machine $sm = (R, r_0, \psi, \chi, E)$ is a sequence of transitions such that

- $t_1 = (s_0, l, s')$, i.e., the path starts in the initial state s_0 of region r, and
- $\forall t_i = (s_{i-1}, l_i, s_i), 1 \leq i < k$: $t_{i+1} = (s_i, l_{i+1}, s_{i+1})$, i.e., the target state of a transition is the source state of the subsequent transition

holds.

Based on the definition of a region path, state machine steps, and state machine runs, a *state machine path* is specified by a sequence of sets of transitions taking the hierarchy and concurrency of regions into account. Therefore, the path starts in the initial state of the root region or in one of its subregions and subsequent transitions are either contained (1) in the same region, (2) in a concurrent region, (3) in a subregion, or (4) in a parent region. The sequence of the sets of transitions is defined by a given state machine run *smr* for the state machine *sm*. Furthermore, it must hold that the projection of transitions of the state machine path on their containing regions define every time a valid region path with $\rho_r = (t_1, \ldots, t_k)$.

Definition 3.7: State Machine Path

Let T_R^* be the set of all paths of a state machine *sm* defined over the set $\mathcal{SMR}_{sm}$ of state machine runs. By the function proj_r, we refer to the projection of a state machine path to a specific region. For a given state machine run $smr = (st_0, \ldots, st_l)$ of a state machine *sm* a *state machine path* $\rho_{sm} = (T_0, \ldots, T_k) \in T_R^*$ of length l is a sequence of sets of transitions such that

- $T_i = \{t, \ldots, t'\} = T_{st_i}, 0 \leq i \leq l$, i.e., every set of transitions of the state machine path corresponds to a certain set of transitions of the state machine run,
- $\forall t = (s, l, s') \in T_0 : s = s_0^r \wedge (r = r_0 \vee r \in \psi^*(s_0^{r_0}))$, i.e., the path starts in the initial state s_0 of the root region r_0 or in the initial states $s_0 = s_0^r$ of the subregions $r \in \psi^*(s_0^{r_0})$ of $s_0^{r_0}$,
- $\forall r \in R : \text{proj}_r(\rho_{sm}) = \rho_r$, i.e., the projection of the state machine path to a specific region denotes a valid region path.

Based on those definitions, a state machine is defined as well-formed if it fulfills the following three properties as similar defined in the literature [UL06; UPL12; LPK+14]. First, every state has to be *reachable* via a state machine path, i.e., every abstract computational state of the system to be modeled can come into effect. Second, the state machine has to be *connected*, i.e., the state machine defines a rooted tree to specify the complete behavior of the system. Third, the state machine has to be *deterministic*, i.e., for every input event sent by the environment, the reaction of the system represented by the state machine is clearly specified. We require well-formed state ma-

chines to apply the formalism for test-modeling purposes, e.g., to allow for valid test-case generation [UL06; UPL12; LPK+14] or to facilitate change impact analysis between variants and versions of variants (cf. Chapt. 4). In the remainder of this thesis, we use the term state machine, but assume every state machine to be well-formed.

Definition 3.8: Well-formed State Machine

A state machine $sm = (R, r_0, \psi, \chi, E)$ is *well-formed*, if the following holds:

- Every state $s \in S_R$ is *reachable* via a path $\rho_{sm} = (T_0, \ldots, T_k) \in T_R^*$ with $\exists t = (s', l, s) \in T_k$.
- Every region $r \in R \setminus \{r_0\}$ is connected to the state machine via the hierarchy functions ψ and χ which follows from the definition of state machines (cf. Def. 3.2).
- For every state $s \in S_R$, its leaving transitions $t = (s, l, s'), t' = (s, l', s'') \in T_R, t \neq t'$, have distinct labels in terms of triggering events and, thus, cannot be executed at the same time to be deterministic which follows from the definition of state machine regions (cf. Def. 3.1).

We define the universe of all well-formed state machines which is specified over the universe $\mathcal{E}$ of events by $\mathcal{SM}$.

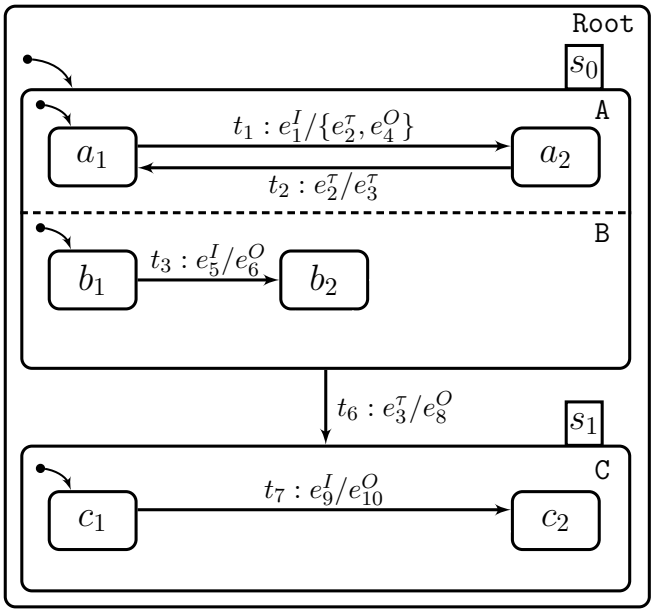


Figure 3.2: Sample State Machine Path ρ_{sm}

Example 3.2: State Machine Execution Semantics

Consider the sample state machine $sm = (R, r_0, \psi, \chi, E)$ in Fig. 3.1 from Ex. 3.1 again. A sample state machine $smr \in \mathcal{SMR}_{sm}$ of sm can be derived as $smr = (st_0, st_1, st_2, st_3, st_4)$, where

$st_0 = (C, E^I_{st_0} T_{st_0}, E^0_{st_0}, C') = (\{(\texttt{Root}, s_0), (\texttt{A}, a_1), (\texttt{B}, b_1)\}, \{e_5^I\}, \{t_3\}, \{e_6^O\}, \{(\texttt{Root}, s_0), (\texttt{A}, a_1), (\texttt{B}, b_2)\})$,

$st_1 = (\{(\texttt{Root}, s_0), (\texttt{A}, a_1), (\texttt{B}, b_2)\}, \{e_1^I\}, \{t_1\}, \{e_2^\tau, e_4^O\}, \{(\texttt{Root}, s_0), (\texttt{A}, a_2), (\texttt{B}, b_2)\})$,

$st_2 = (\{(\texttt{Root}, s_0), (\texttt{A}, a_2), (\texttt{B}, b_2)\}, \{e_2^\tau\}, \{t_2\}, \{e_3^\tau\}, \{(\texttt{Root}, s_0), (\texttt{A}, a_1), (\texttt{B}, b_2)\})$,

$st_3 = (\{(\texttt{Root}, s_0), (\texttt{A}, a_1), (\texttt{B}, b_2)\}, \{e_3^\tau\}, \{t_6\}, \{e_8^O\}, \{(\texttt{Root}, s_1), (\texttt{C}, c_1)\})$, and
$st_4 = (\{(\texttt{Root}, s_1), (\texttt{C}, c_1)\}, \{e_9^I\}, \{t_7\}, \{e_{10}^O\}, \{(\texttt{Root}, s_1), (\texttt{C}, c_2)\})$
holds. The sample state machine run results in the sample state machine path $\rho_{sm} = (\{t_3\}, \{t_1\}, \{t_2\}, \{t_6\}, \{t_7\})$ shown in Fig. 3.2 starting in the initial state s_0 of the root region and due to its hierarchical composition also in the initial state a_1 of subregion A and b_1 of subregion B. The path ρ_{sm} ends in state c_2 of region C which is a subregion of state s_1 of the root region. In addition, the state machine region path $\rho_r^A = \mathsf{proj}_A(\rho_{sm}) = (t_1, t_2)$ for region A is derivable.

Table 3.1: Symbol Summary of State Machine Definition

Symbol	Description
$s; S; \mathcal{S}$	Abstract computation state; Finite set of states; Universe of states
$t; T; \mathcal{T}; T^*$	Transition; Finite set of transitions; Universe of transitions; Sequence of transitions
$e; E; \mathcal{E}$	Event; Finite set of events; Universe of events
$l; L; \mathcal{L}$	Transition label; Finite set of transition labels; Universe of transitions labels
$r; R; \mathcal{R}$	State machine region; Finite set of regions; Universe of regions
ψ	Sub-hierarchy function
χ	Parent-hierarchy function
$sm; SM; \mathcal{SM}$	State machine; Finite set of state machines; Universe of state machines
ρ_r	State machine region path
ρ_{sm}	State machine path
proj_r	State machine path projection function
$C; \mathcal{C}_{sm}$	State machine configuration; Set of state machine configurations
$st; \mathcal{ST}_{sm}$	State machine step; Set of state machine steps
$smr; \mathcal{SMR}_{sm}$	State machine run; Set of state machine runs

The list of symbols used for the definition of our state machine dialect is summarized in Tab. 3.1. To recapitulate, a state machine $sm = (R, r_0, \psi, \chi, E)$ is specified by a set R of regions, a designated root region r_0, the hierarchy functions ψ and χ, and a set E of events. In addition, a region $r = (S, s_0, E, L, T)$ comprises a set S of abstract computation states, an initial state s_0, a set E of events, a set L of transition labels, and a set T of transitions. We are able to derive a state machine path ρ_{sm} denoting a potential execution of the system as defined by a state machine run *smr*. Each run *smr* is a sequence of state machine steps st_i specifying the transfer from one state machine configuration C to the next one C'. Besides state machine paths, we are also able to focus solely on a path ρ_r within a specific region r by applying the state machine path projection function proj_r on ρ_{sm}.

3.2 Delta-Oriented Test Modeling for Variants

Variability implementation techniques [SRC+12] can be defined as generically applicable and, thus, independent from a concrete domain artifact type. Hence, for a specific artifact type and usage scenario, those techniques have to be instantiated. In this thesis, we propose model-based regression

testing for evolving product lines, where we use state machines for test-modeling purposes as common for model-based testing [UL06; UPL12]. We instantiate and describe delta modeling [CHS15; Sch10] accordingly in this section. We focus on delta modeling as it is already applied in the context of efficient SPL testing [LSK+12; LLS+12; Loc13; LLL+14; LLL+15; VBM15; LLA+16; LMT+16; DFG+17; LNT+19] and, therefore, builds the foundations for our extension to facilitate the specification of the behavior of evolving SPLs (cf. Sect. 3.3).

Delta State Machines

The transformational variability implementation technique delta modeling [Sch10; SBB+10; SD10; CHS15] is a flexible and modular approach for the development of variable domain artifacts and, hence, for the development of SPLs, where the differences between variants are explicitly captured as transformations encapsulated in deltas. Delta modeling [Sch10; CHS15] is already adapted for various types of domain artifacts like software architectures [HKR+11b; HKR+11a; LLL+14; LLL+15], fault diagrams [SSA13b], requirements [DSL+13], Matlab/Simulink [HKM+13], code [SBB+10; SD10; KHS+14], class diagrams [Sch10], performance-annotated activity diagrams [KTT+15], and the process calculus CCS [LMB+14; LMB+16]. Furthermore, the application scenarios vary from incremental testing [LSK+12; LLS+12; DSL+13; LLL+14; LLL+15; LLA+16; LBL+17; LMT+16; VBM15; DFG+17; LNT+19] over safety analysis [SSA13b] and verification [BKS11; LMB+14; LMB+16] to (model-driven) development [SBB+10; SD10; Sch10; KHS+14; HKR+11b; HKR+11a; HKM+13; KTT+15] of SPLs. The adaptation of delta modeling [Sch10; CHS15] for a specific artifact type is realized either manually with a respective engineering and tooling effort, or automatically based on a given language grammar [HHK+13; HHK+15] or meta model [SSA14a; PKK+15; CDD+16; PRK+17]. In this thesis, we focus on the adaptation for state machines called delta state machines already applied as test-modeling formalism for incremental model-based SPL testing [LSK+12; LLS+12; LLL+14; LMT+16; LNT+19].

In the original definition of delta modeling, also known as *core delta programming/modeling* [SBB+10; Sch10], a variant-specific model, i.e., a state machine $sm_{v_i} \in SM_{\mathbb{V}}$, is generated by transforming a predefined *core model* $sm_{v_{core}} \in SM_{\mathbb{V}}$ via change operations encapsulated in deltas. By $SM_{\mathbb{V}} \subset \mathcal{SM}$, we refer to the set of variant-specific state machines of an SPL under consideration. In contrast to core delta modeling [SBB+10; Sch10], in *pure delta programming/modeling* [SD10], there is no designated core model and, therefore, a variant-specific model is generated by applying deltas on an empty model $sm_{\emptyset} \in \mathcal{SM}$. According to Schaefer and Damiani [SD10], the pure version of delta modeling facilitates an easier handling of SPL evolution as solely deltas have to be modified, added or removed to correspond to the new SPL version. Despite this, core delta modeling is more suitable for incremental SPL testing as it allows to use the core as starting point of the test process to increase the reuse potential of test artifacts for the remaining variants to be tested [LLL+14; LLL+15; LNT+19]. However, both strategies are equivalent in terms of the resulting variant-specific models and, therefore, are transformable into each other based on a respective encoding [SD10]. In this thesis, we apply core delta modeling [SBB+10; Sch10] as foundation for our extension to capture the variability as well as evolution of variants (cf. Sect. 3.3), and for the definition of our regression testing framework (cf. Chapt. 5). Moreover, based on the encoding of core into pure delta modeling [SD10; SBB+10], our extension of delta modeling is also applicable for pure delta modeling [SD10].

For the selection of the core, several alternatives exist. By choosing one of the smallest variants as core, the required time for its testing or development gets potentially decreased. For efficient

SPL testing, the selection of the variant as core (1) which comprises the most commonality shared between all variants in its variant-specific model facilitates the exploitation of the existing reuse potential of test artifacts [LLL+14; LLL+15; LNT+19], or (2) which is the largest in terms of number of features or model elements allows for an increased test coverage at the beginning of the test process [ATL+16; ALL+17]. In contrast, business decisions also may influence the selection of the core, e.g., the variant which should be sold as first variant could be used as core. In this thesis, we choose the variant $v_{core} \in \mathbb{V}$ as core which shares the most commonality between all variants in its state machine $sm_{v_{core}} \in SM_{\mathbb{V}}$. That means, if we would compare all variant-specific state machines against each other and sum up their differences in terms of state machine elements, the most common variant has the smallest number of overall differences. By selecting the most common variant as core, we are able to increase the efficiency of our regression testing framework (cf. Chapt. 5) even though our framework also handles a core obtained by one of the other selection strategies.

The selection of the core v_{core} and its state machine $sm_{v_{core}}$ influences the specification of deltas. A *state machine delta* encapsulates change operations to transform $sm_{v_{core}}$ into a variant-specific state machine $sm_{v_i} \in SM_{\mathbb{V}}$ for a variant $v_i \in \mathbb{V}$. For state machines, a *change operation* either (1) adds or removes states, (2) adds or removes transitions, (3) adds or removes regions, (4) adds or removes a subregion hierarchy relation, or (5) modifies a transition label. For the addition of regions, we incorporate completely defined regions and partially defined regions. A partially defined region allows for the addition of further states and transitions via applicable change operations. We similarly incorporate completely and partially defined regions for the removal change operation such that further states and/or transitions may have to be removed by additional change operations.

Definition 3.9: State Machine Change Operation

Let $OP \subset \mathcal{OP}$ be the set of all change operations defined over the elements of the current SPL under consideration which is further a subset of the universe of change operations defined over the universes of states $\mathcal{S}$, transitions $\mathcal{T}$, transition labels $\mathcal{L}$, and regions $\mathcal{R}$. A *state machine change operation* $op \in OP$ defines one of the following transformations:

- add s, i.e., a state $s \in \mathcal{S}$ is added,
- rem s, i.e., a state $s \in \mathcal{S}$ is removed,
- add t, i.e., a transition $t \in \mathcal{T}$ is added,
- rem t, i.e., a transition $t \in \mathcal{T}$ is removed,
- mod(t, l'), i.e., the label of transition $t = (s, l, s')$ is exchanged by $l' \in \mathcal{L}$,
- add r, i.e., a region $r \in \mathcal{R}$ is added,
- rem r, i.e., a region $r \in \mathcal{R}$ is removed,
- add (s, r), i.e., a subregion hierarchy relation is defined between state $s \in \mathcal{S}$ and region $r \in \mathcal{R}$, or
- rem (s, r), i.e., a subregion hierarchy relation between state $s \in \mathcal{S}$ and region $r \in \mathcal{R}$ is removed.

We do not allow for the manipulation of initial states by applying change operations. Hence, the initial states defined in the core state machine or defined in regions to be added are immutable during the transformation process. As states and transitions are contained in a specific region, we require the information for which region a change operation, e.g., state addition, has to be applied.

The same holds for the addition and removal of a subregion hierarchy relation as we require the information in which region the respective state is comprised. In contrast, the addition and removal of regions does not require such information as a region is captured directly by a state machine. Therefore, a delta defines for which region the captured change operations has to be applied. The designated region is either part of the core state machine $sm_{v_{core}}$ or is part of an intermediate state machine sm obtained from the incremental application of deltas, i.e., the region is added by another delta which is applied before the delta that captures the corresponding change operation. Furthermore, a delta comprises a Boolean expression $\mathbb{B}(F)$ over features as *application condition* that does not violate the constraints of the respective feature model *fm*. The application condition specifies for which feature (configuration) the delta has to be applied to transform the core $sm_{v_{core}}$.

Definition 3.10: State Machine Delta

A *state machine delta* $\delta = (OP_\delta, r_\epsilon, \varphi_\delta)$ is defined as triple with

- $OP_\delta = \{op_1, \ldots, op_m\} \subseteq OP$ is a finite set of *change operations*, where further holds that
  - $\forall op = \mathsf{mod}(t, l') \in OP_\delta : \neg\exists op' = \mathsf{mod}(t, l'')$, i.e., the set of change operations contains at most one modification of a transition label for a transition t
  - $\forall op = \mathsf{add}\, s \in OP_\delta : \neg\exists op' = \mathsf{rem}\, s$, i.e., the set of change operations does not contain an addition and removal of the same state
  - $\forall op = \mathsf{add}\, t \in OP_\delta : \neg\exists op' = \mathsf{rem}\, t$, i.e., the set of change operations does not contain an addition and removal of the same transition
  - $\forall op = \mathsf{add}\, r \in OP_\delta : \neg\exists op' = \mathsf{rem}\, r$, i.e., the set of change operations does not contain an addition and removal of the same region, and
  - $\forall op = \mathsf{add}\,(s, r) \in OP_\delta : \neg\exists op' = \mathsf{rem}\,(s, r)$, i.e., the set of change operations does not contain an addition and removal of the same subregion hierarchy relation.
- $r_\epsilon \in \mathcal{R} \cup \{\epsilon\}$ is a region $r \in \mathcal{R}$ the change operations are applied to or ϵ denoting that the information is not required, and
- $\varphi_\delta \in \mathbb{B}(F)$ is an application condition corresponding to the feature model *fm* such that

$$\exists F_{v_i} \in F_{\mathbb{V}} : [\![F_{v_i}]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta$$

holds, i.e., there exists at least one variant-specific feature configuration F_{v_i} that satisfies the feature model *fm* and the application condition φ_δ. By $[\![.]\!] : F_{\mathbb{V}} \to (F \to \mathbb{B})$ with $F_{\mathbb{V}} \subseteq 2^F$, we define a *feature selection function* converting a subset of features $F_{v_i} \in F_{\mathbb{V}}$ into a function that provides a Boolean valuation for features $f \in F$ used as variables in $[\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta$ to reason about the satisfiability such that the following holds

$$\forall f \in F : [\![F_{v_i}]\!](f) = \begin{cases} \texttt{true} & \text{if } f \in F_{v_i} \\ \texttt{false} & \text{otherwise} \end{cases}$$

The correspondence between feature model and application conditions is specified by the configuration knowledge [SRC+12; CE00]. We abstract from a certain formalism for the configuration knowledge and assume the valid mapping between problem space and solution space to be given. For each SPL represented by its set of variants $\mathbb{V}$, a set of state machine deltas is composed to trans-

form the core state machine $sm_{v_{core}}$ into all other variant-specific state machines $sm_{v_i} \in SM_{\mathbb{V}}$ for variants $v_i \in \mathbb{V}$. Therefore, we capture both as *state machine delta model* of an SPL.

Definition 3.11: State Machine Delta Model

A *delta model* $DM = (sm_{v_{core}}, \Delta_{DM})$ is defined as tuple, where

- $sm_{v_{core}} \in SM_{\mathbb{V}}$ is the core state machine of the core $v_{core} \in \mathbb{V}$ to be transformed, and
- $\Delta_{DM} = \{\delta_1, \ldots, \delta_m\}$ is a finite set of state machines deltas to transform $sm_{v_{core}}$.

We define $\mathcal{DM}$ as the set of all possible delta models defined over $\mathcal{OP}$.

In practice, a delta model can be created (1) manually with respective tool support following a proactive modeling process [Kru02; CDD+16], (2) automatically using model differencing techniques [PKK+15; PRK+17] allowing for proactive, reactive, and extractive modeling [Kru02], or (3) automatically using family mining techniques [WRS+17] allowing for delta model extraction. For our SPL regression testing framework (cf. Chapt. 5), we abstract from the concrete creation, but we assume a delta model as given denoting the variable behavioral specification of an SPL under test.

Delta Application. To transform core state machine $sm_{v_{core}}$ into a variant-specific state machine sm_{v_i}, we first determine the set of deltas to be applied. Based on a feature configuration $F_{v_i} \in F_{\mathbb{V}}$ of a variant $v_i \in \mathbb{V}$, we evaluate for each delta $\delta_j \in \Delta_{DM}$ if its application condition φ_{δ_j} is satisfied $[\![F_{v_i}]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_{\delta_j}$ or not $[\![F_{v_i}]\!] \not\models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_{\delta_j}$. Hence, for each variant $v_i \in \mathbb{V}$ a delta (sub)set $\Delta_{v_i} \subseteq \Delta_{DM}$ exists solely comprising deltas $\delta_j \in \Delta_{DM}$ with satisfied application condition. A determined delta set Δ_{v_i} is incrementally applied to the core $sm_{v_{core}}$ to obtain state machine sm_{v_i} of variant v_i. The incremental application has to guarantee that the result is a unique and well-formed variant-specific state machine. Hence, there exist some prerequisites for the application as follows.

- A state machine element, e.g., a state, can only be added, if it does not already exist in the (intermediate) state machine to be transformed. Furthermore, for the addition of states and transitions, the region r_ϵ to be extended has to exist. The same holds for the addition of a subregion hierarchy relation, where the state as well as the subregion to be related has to be contained in the (intermediate) state machine. This is the case, if either the elements are part of the core state machine or are added by another delta which has to be applied in advance.
- A state machine element, e.g., a region, can only be removed, if it exists in the (intermediate) state machine to be transformed.
- A transition can only be modified, if it exists in the (intermediate) state machine to be transformed.
- The union of change operations of the set of deltas to be applied to the core state machine does not contain an addition and a removal of the same element, e.g., state.
- The union of change operations of the set of deltas to be applied to the core state machine does not contain several modifications of the transition label of the same transition.

To ensure those prerequisites, we can exploit type systems for delta-oriented SPLs [DS12; DL16; LDT+18]. Variant-specific delta sets Δ_{v_i} can further be analyzed and ordered as sequence to be applied on the core $sm_{v_{core}}$. As not every delta $\delta \in \Delta_{v_i}$ is dependent on the application of another delta $\delta' \in \Delta_{v_i}$, the sequence is derivable as partial order. If such an application sequence does not exist, the delta model is not valid and should be corrected. We refer to Clarke et al. [CHS15] for a detailed discussion about delta and change operation conflicts and how they can be handled or solved.

We define the incremental application of a variant-specific delta set based on the function apply_δ. The function takes the core state machine $sm_{v_{core}}$ as well as a delta set Δ_v as input and determines as first step the application sequence of the deltas $\delta_j \in \Delta_v$ by analyzing the encapsulated change operations. Afterwards, the sequence of deltas is applied until no delta remains in the sequence, where as a second step, the function apply_δ also determines, for each delta to be applied, the application sequence of its captured change operations. The resulting sequence of change operations is defined as follows:

1. Removal of transitions
2. Removal of subregion hierarchy relations
3. Removal of states
4. Removal of regions
5. Addition of regions
6. Addition of states
7. Addition of subregion hierarchy relations
8. Addition of transitions
9. Modification of transitions

Definition 3.12: Delta Application

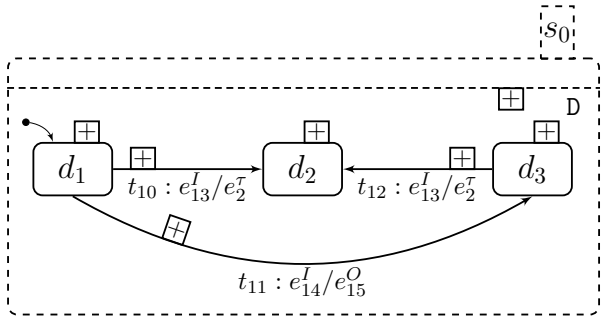
The *incremental application of deltas* is defined via the function

$$\text{apply}_\delta : SM_{\mathbb{V}} \times \mathcal{P}(\Delta_{DM}) \to SM_{\mathbb{V}}$$

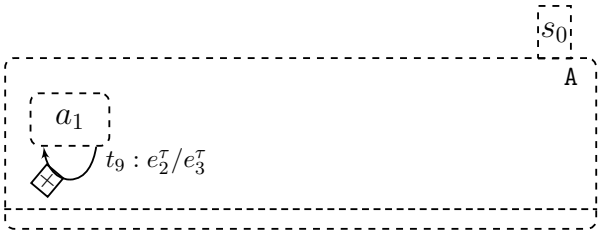
as follows:

- $\text{apply}_\delta(sm, \emptyset) = sm$
- $\text{apply}_\delta(sm, \Delta_v) = \text{apply}_\delta(sm, (\delta_1, \ldots, \delta_m))$
- $\text{apply}_\delta(sm, (\delta_1, \ldots, \delta_j)) = \text{apply}_\delta(\text{apply}_\delta(sm, \delta_1), (\delta_2, \ldots, \delta_j))$
- $\text{apply}_\delta(sm, \delta) = \text{apply}_\delta(sm, \{op_1^\delta, \ldots, op_m^\delta\})$
- $\text{apply}_\delta(sm, \{op_1^\delta, \ldots, op_m^\delta\}) = \text{apply}_\delta(sm, (op_1^\delta, \ldots, op_m^\delta))$
- $\text{apply}_\delta(sm, (op_1^\delta, \ldots, op_m^\delta)) = \text{apply}_\delta(\text{apply}_\delta(sm, op_1^\delta), (op_2, \ldots, op_m))$
- $\text{apply}_\delta(sm, \text{add}\, s) = sm' = (R', r_0, \psi, E)$ with $R' = (R \setminus \{r_\epsilon\}) \cup \{r'_\epsilon\}$ and $r'_\epsilon = (S \cup \{s\}, s_0, E, T, L)$
- $\text{apply}_\delta(sm, \text{rem}\, s) = sm' = (R', r_0, \psi, E)$ with $R' = (R \setminus \{r_\epsilon\}) \cup \{r'_\epsilon\}$ and $r'_\epsilon = (S \setminus \{s\}, s_0, E, T, L)$
- $\text{apply}_\delta(sm, \text{add}\, t) = sm' = (R', r_0, \psi, E)$ with $R' = (R \setminus \{r_\epsilon\}) \cup \{r'_\epsilon\}$ and $r'_\epsilon = (S, s_0, E, T \cup \{t\}, L)$
- $\text{apply}_\delta(sm, \text{rem}\, t) = sm' = (R', r_0, \psi, E)$ with $R' = (R \setminus \{r_\epsilon\}) \cup \{r'_\epsilon\}$ and $r'_\epsilon = (S, s_0, E, T \setminus \{t\}, L)$
- $\text{apply}_\delta(sm, \text{mod}(t, l')) = sm' = (R', r_0, \psi, E)$ with $R' = (R \setminus \{r_\epsilon\}) \cup \{r'_\epsilon\}$ and $r'_\epsilon = (S, s_0, E, (T \setminus \{t = (s, l, s')\}) \cup \{t' = (s, l', s')\}, L)$
- $\text{apply}_\delta(sm, \text{add}\, r) = sm' = (R \cup \{r\}, r_0, \psi, E)$
- $\text{apply}_\delta(sm, \text{rem}\, r) = sm' = (R \setminus \{r\}, r_0, \psi, E)$
- $\text{apply}_\delta(sm, \text{add}\, (s, r)) = sm' = (R, r_0, \psi \cup \{(s, r)\}, E)$
- $\text{apply}_\delta(sm, \text{rem}\, (s, r)) = sm' = (R, r_0, \psi \setminus \{(s, r)\}, E)$

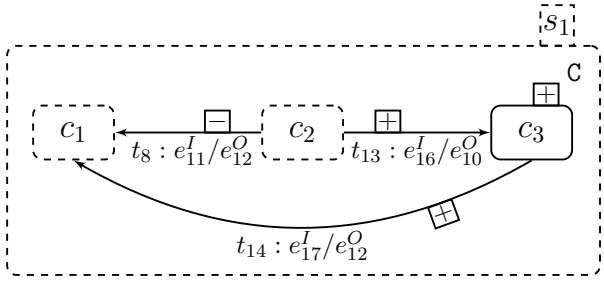
The successful application of a variant-specific delta set results in a unique and well-formed state machine. However, due to the incrementality, intermediate state machines may temporarily violate the well-formedness conditions, e.g., a state is added but not yet connected. Furthermore, due to the addition and removal of transitions and regions, the set of events E_{r_ϵ} of the region $r_\epsilon \in R_{sm}$ as well as the set of events $E_{sm} = \bigcup_{r \in R_{sm}} E_r$ of the (intermediate) state machine *sm* to be transformed are automatically updated at the end of the state machine delta application. Hence, in the end, the event sets correspond to the events used in transition labels.



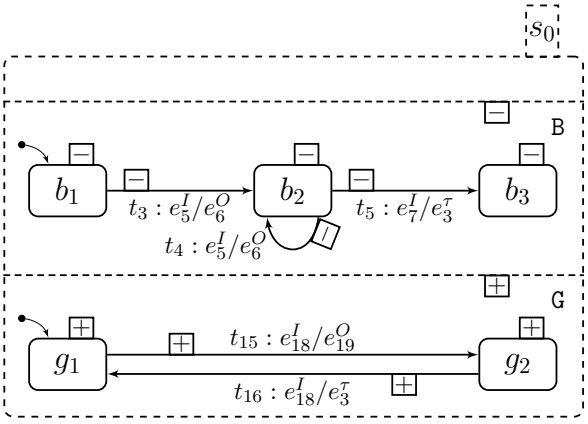
(a) δ_1 with $r_\epsilon =$ Root and $\varphi_{\delta_1} = f_1$



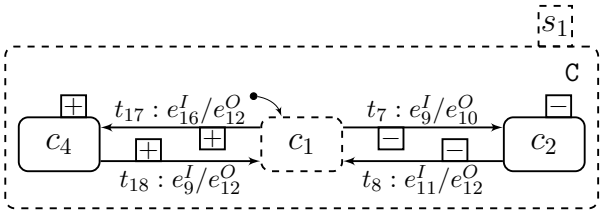
(b) δ_2 with $r_\epsilon =$ A and $\varphi_{\delta_2} = f_1$



(c) δ_3 with $r_\epsilon =$ C and $\varphi_{\delta_3} = f_2 \wedge f_1$



(d) δ_4 with $r_\epsilon =$ Root and $\varphi_{\delta_4} = f_3$



(e) δ_5 with $r_\epsilon =$ C and $\varphi_{\delta_5} = f_3 \wedge f_1$

Figure 3.3: Sample Delta Set Δ_{DM}

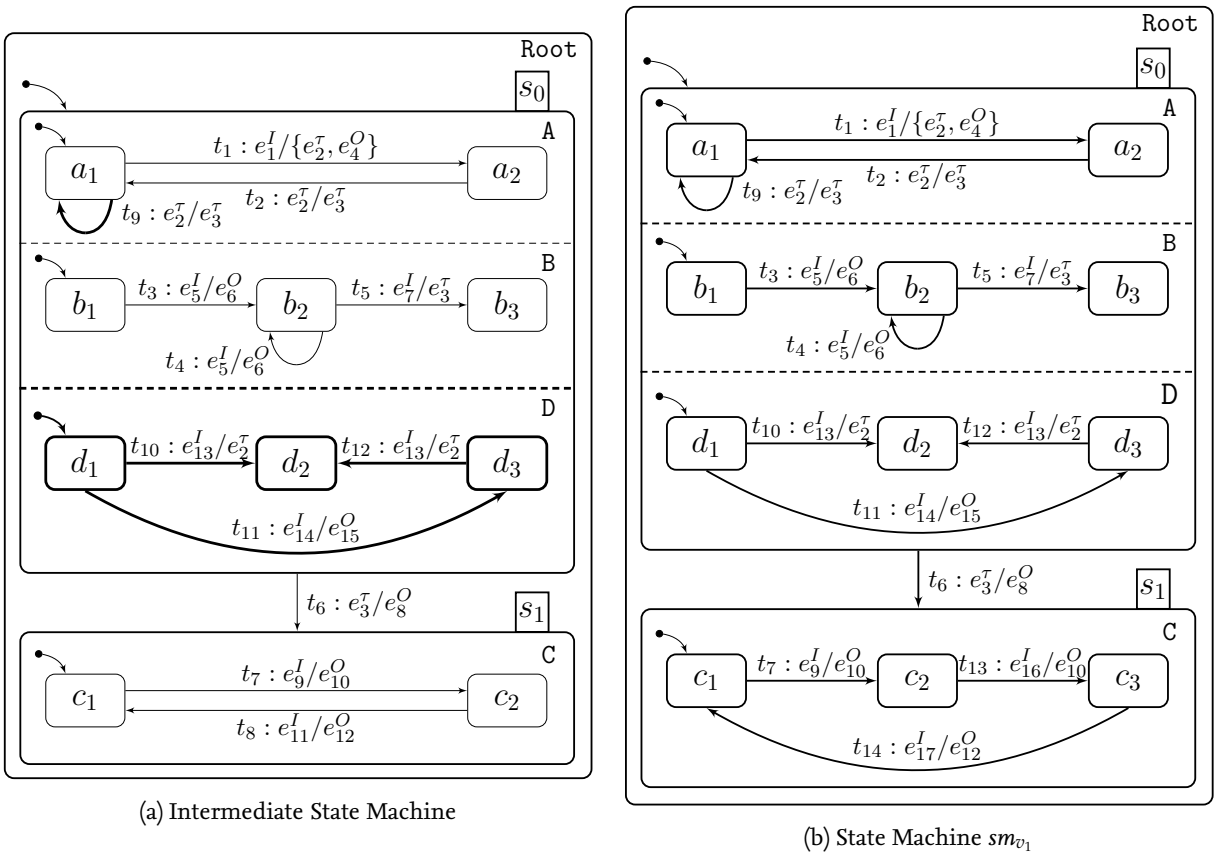
Example 3.3: Delta Modeling

Consider the state machine *sm* from Ex. 3.1 shown in Fig. 3.1 again, now used as core state machine $sm_{v_{core}}$ of the variant $v_{core} = v_0 \in \mathbb{V}$ from Ex. 2.1. For the generation of the state machines sm_{v_i} of the other variants $v_i \in \mathbb{V} \setminus \{v_{core}\}$, we define four deltas depicted in Fig. 3.3

as follows, where the feature model *fm* to specify application conditions is shown in Fig. 2.5:

$\delta_1 = (OP, r_e, \varphi) = (\{\mathsf{add}\,(s_0, \mathtt{D}), \mathsf{add}\,\mathtt{D}\}, \mathtt{Root}, f_1)$
$\delta_2 = (\{\mathsf{add}\,t_9\}, \mathtt{A}, f_1)$
$\delta_3 = (\{\mathsf{rem}\,t_8, \mathsf{add}\,c_3, \mathsf{add}\,t_{13}, \mathsf{add}\,t_{14}\}, \mathtt{C}, f_2 \wedge f_1)$
$\delta_4 = (\{\mathsf{rem}\,(s_0, \mathtt{B}), \mathsf{rem}\,\mathtt{B}, \mathsf{add}\,\mathtt{G}, \mathsf{add}\,(s_0, \mathtt{G})\}, \mathtt{Root}, f_3)$
$\delta_5 = (\{\mathsf{rem}\,t_7, \mathsf{rem}\,t_8, \mathsf{rem}\,c_2, \mathsf{add}\,c_4, \mathsf{add}\,t_{17}, \mathsf{add}\,t_{18}\}, \mathtt{C}, f_3 \wedge f_1)$

We combine $sm_{v_{core}}$ and $\Delta_{DM} = \{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5\}$ in a delta model $DM = (sm_{v_{core}}, \Delta_{DM})$. To obtain, e.g., the state machine sm_{v_1} of variant $v_1 \in \mathbb{V}$, we determine the variant-specific delta set $\Delta_{v_1} = \{\delta_1, \delta_2, \delta_3\}$ by evaluating and selecting a delta from Δ_{DM} if its application condition is satisfied by the feature configuration $F_{v_1} \in F_{\mathbb{V}}$, i.e., $[\![F_{v_i}]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta$. The incremental application of $\Delta_{v_1} = \{\delta_1, \delta_2, \delta_3\}$ results in the state machine sm_{v_1} depicted in Fig. 3.4b, i.e., $sm_{v_1} = \mathsf{apply}_\delta(sm_{v_{core}}, \Delta_{v_1})$. The respective intermediate state machine after applying δ_1 and δ_2 is shown in Fig. 3.4a, where the differences to the core are highlighted.



(a) Intermediate State Machine

(b) State Machine sm_{v_1}

Figure 3.4: Sample Incremental Delta Application

So far, a delta model *DM* solely captures the differences between the core state machine $sm_{v_{core}}$ and other variant-specific state machines sm_{v_i} in terms of deltas. However, for our model-based regression testing framework (cf. Chapt. 5), we are further interested in the differences between arbitrary variants $v_i, v_j \in \mathbb{V}$. To determine those differences, we derive *state machine regression deltas* [LSK+12; LLS+12; Loc13; LLL+14] by combining their delta sets Δ_{v_i} and Δ_{v_j} using an adaptation of the sym-

metric difference operator. Therefore, a state machine regression delta captures and inverts those deltas only applied for v_i and combines them with deltas only applied for v_j. An inversion of a delta $(\delta)^{-1}$ is defined by the inversion $(op)^{-1}$ of its captured change operations $op \in OP_\delta$, where an addition results in a removal, a removal in an addition, and a modification is recovered by exchanging the modified element with its original version. All change operations op as well as their counterparts $(op)^{-1}$ are comprised by definition in the universe $\mathcal{OP}$ of change operations such that an inverted delta always exists. The region r_ϵ^δ still defines the region to be modified by the inverted change operations, but the application condition φ_δ is not incorporated in the regression delta computation and application. We are able to neglect the application conditions for the computation as the application conditions are solely required to determine the variant-specific delta sets Δ_{v_i} and Δ_{v_j} which are already known for the regression delta computation. Hence, the computation can focus on the change operations captured by the deltas of Δ_{v_i} and Δ_{v_j}.

Definition 3.13: State Machine Regression Delta [LSK+12; LLS+12; Loc13; LLL+14]

A *state machine regression delta* $\Delta_{v_i,v_j} = \{\delta_1^{-1}, \ldots, \delta_{n-1}^{-1}, \delta_n, \ldots \delta_m\}$ is derived by

$$\Delta_{v_i,v_j} = (\Delta_{v_i} \setminus \Delta_{v_j})^{-1} \cup (\Delta_{v_j} \setminus \Delta_{v_i})$$

such that $sm_{v_j} = \mathsf{apply}_\delta(sm_{v_i}, \Delta_{v_i,v_j})$ holds.

Based on the delta inversion, we ensure that state machine elements contained in sm_{v_i} and not in sm_{v_j} gets removed and vice versa. For detailed information about the construction of a state machine regression delta and the proof of its existence, we refer the reader to the literature [LSK+12; LLS+12; Loc13; LLL+14]. Similar to Tab. 3.1, we summarize the list of symbols used for the definition of delta state machines in Tab. 3.2. To recapitulate, a state machine delta $\delta = (OP_\delta, r_\epsilon, \varphi_\delta)$ encapsulates a set OP_δ of change operations, e.g., the addition of a state, the region r_ϵ the change operations have to applied to, and the application condition φ_δ specifying for which feature (configuration) the delta has to be applied to transform the core $sm_{v_{core}}$. A delta model $DM = (sm_{v_{core}}, \Delta_{DM})$ is defined by the combination of the core state machine $sm_{v_{core}}$ and a set Δ_{DM} of deltas. To transform the core state machine $sm_{v_{core}}$ into another variant-specific state machine sm_{v_i}, the application condition of each delta is evaluated based on a given feature configuration $F_{v_i} \in F_{\mathbb{V}}$ and the feature selection function $[\![F_{v_i}]\!]$. If the application condition is evaluated to `true`, the respective delta is gathered in the variant-specific delta set Δ_{v_i} which is then incrementally applied to the core using the delta application function apply_δ to obtain the state machine sm_{v_i}. In addition, we are able to determine the differences between two arbitrary variant-specific state machines sm_{v_i} and sm_{v_j} captured in a state machine regression delta Δ_{v_i,v_j} by taking their delta sets Δ_{v_i} and Δ_{v_j} into account.

Example 3.4: State Machine Regression Delta

Consider the sample state machine sm_{v_2} for variant v_2 depicted in Fig. 3.5. We either generate the state machine by transforming the core $sm_{v_{core}}$ using the delta in the respective variant-specific delta set $\Delta_{v_2} = \{\delta_4\}$ or by computing the state machine regression delta Δ_{v_1,v_2}

$$= (\Delta_{v_1} \setminus \Delta_{v_2})^{-1} \cup (\Delta_{v_2} \setminus \Delta_{v_1})$$
$$= (\{\delta_1, \delta_2, \delta_3\} \setminus \{\delta_4\})^{-1} \cup (\{\delta_4\} \setminus \{\delta_1, \delta_2, \delta_3\})$$

$= (\{\delta_1, \delta_2, \delta_3\})^{-1} \cup \{\delta_4\}$
$= \{\delta_1^{-1}, \delta_2^{-1}, \delta_3^{-1}, \delta_4\}$
$= \{(OP_{\delta_1}, r_\epsilon^{\delta_1}, \varphi_{\delta_1})^{-1}, (OP_{\delta_2}, r_\epsilon^{\delta_2}, \varphi_{\delta_2})^{-1}, (OP_{\delta_3}, r_\epsilon^{\delta_3}, \varphi_{\delta_3})^{-1}, \delta_4\}$
$= \{((\{\mathsf{add}\,(s_0, \mathtt{D}), \mathsf{add}\,\mathtt{D}\})^{-1}, r_\epsilon^{\delta_1}, \varphi_{\delta_1}), ((\{\mathsf{add}\,t_9\})^{-1}, r_\epsilon^{\delta_2}, \varphi_{\delta_2}), ((\{\mathsf{rem}\,t_8, \mathsf{add}\,c_3, \mathsf{add}\,t_{13}, \mathsf{add}\,t_{14}\})^{-1}, r_\epsilon^{\delta_3}, \varphi_{\delta_3}), \delta_4\}$
$= \{(\{\mathsf{rem}\,(s_0, \mathtt{D}), \mathsf{rem}\,\mathtt{D}\}, r_\epsilon^{\delta_1}, \varphi_{\delta_1}), (\{\mathsf{rem}\,t_9\}, r_\epsilon^{\delta_2}, \varphi_{\delta_2}), (\{\mathsf{add}\,t_8, \mathsf{rem}\,c_3, \mathsf{rem}\,t_{13}, \mathsf{rem}\,t_{14}\}, r_\epsilon^{\delta_3}, \varphi_{\delta_3}), \delta_4\}$

and applying it on state machine sm_{v_1}.

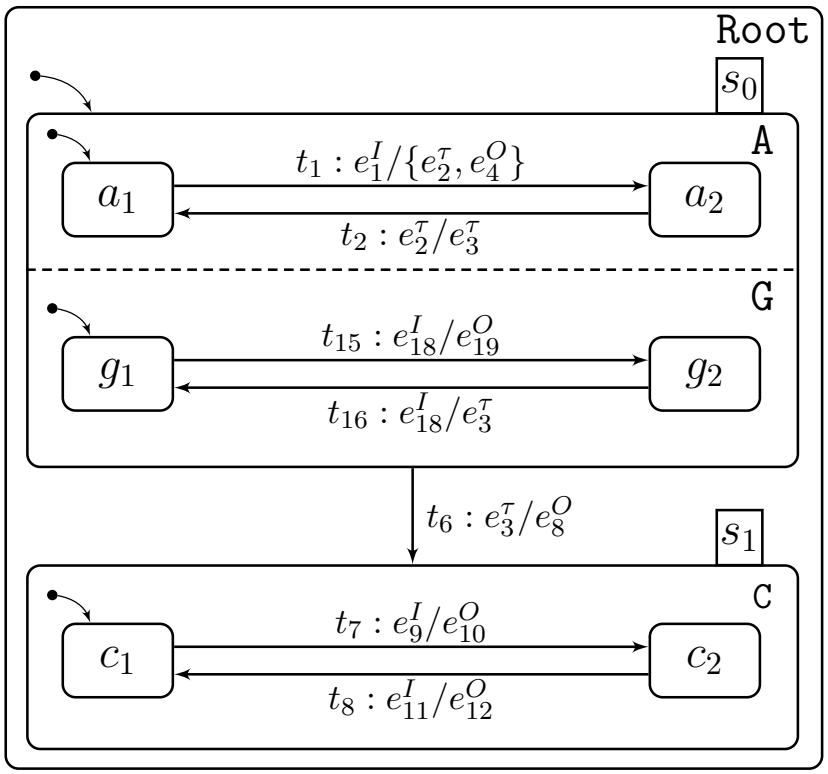


Figure 3.5: State Machine sm_{v_2}

Table 3.2: Symbol Summary of Delta State Machine Definition

Symbol	Description
$op; OP; \mathcal{OP}$	Change operation; Finite set of change operation; Universe of change operations
φ	Application condition
$\delta; \Delta$	State machine delta; Finite set of state machine deltas
Δ_{v_i,v_j}	State machine regression delta
$DM; \mathcal{DM}$	State machine delta model; Universe of state machine delta models
$[\![.]\!]$	Feature selection function
apply_δ	Delta application function

Delta state machines capture the variability of variants of one SPL version in time. To also incorporate the variability of versions of variants and, hence, to take SPL evolution into account for variability modeling in the solution space, we propose an extension of delta modeling [Sch10; CHS15] called higher-order delta modeling in the next section, where the variability and the version information of variants are considered as first-class entities. We apply higher-order delta modeling as test-modeling formalism incorporated in our model-based regression testing framework (cf. Chapt. 5).

3.3 Delta-Oriented Test Modeling for Versions of Variants

Due to the increasing longevity of software systems and, hence, also of SPLs, evolution is an inevitable process to react on, e.g., changing requirements or customer requests [SB99; SV02; MSC14; BP14]. The changes to be made for SPL evolution affect artifacts both from the problem space [BPD+10; SSA13a; BKL+16; NSS16], e.g., the feature model, as well as the solution space [ST00; ALR+05; AMC+07; DGR+10; HRR+12; SSA13a; LSK+13; KLL+14; NBA+15; LNT+18], e.g., the code base. In this thesis, we abstract from problem space evolution and focus on solution space evolution as we propose a model-based regression testing framework for testing variants and versions of variants (cf. Chapt. 5) based on their variable behavioral specification. However, we assume that the feature model evolution is applied and managed correctly and refer to the literature for respective techniques [BPD+10; PBD+12; SSA13a; NSS16].

In the previous section, we described the adaptation of delta modeling [Sch10; CHS15] for state machines used as test-modeling formalism for one SPL *version* θ_j of the complete SPL *evolution history* $\Theta = \{\theta_0, ..., \theta_n\}$. The version set Θ of an SPL is defined as an index set and can be instantiated with timestamps, version numbers etc. usable for providing an ordering. In this thesis, each version θ_j represents an *evolution step* of the evolution history starting in the initial version θ_0 up to the present version θ_n. We assume the evolution history to be sequential and abstract from a branching history. Hence, we define the evolution history $(\Theta, <)$ as strict total order caused by consecutive evolution steps. However, to incorporate also branches, we can interpret each branch as a separate sequential evolution history. We write Θ instead of $(\Theta, <)$ for short in the remainder of this thesis.

To take also evolution during test modeling into account, a corresponding modeling technique has to capture the variability as well as version information of variants and their versions. The incorporation of both information further facilitates the application of techniques for analyzing certain aspects of SPL evolution, e.g., to guide SPL regression testing based on change impact analysis (cf. Chapt. 4). Hence, the definition of a test-modeling formalism for evolving SPLs, where variability and versions are handled as first-class entities, supports the test process to be more efficient. Existing techniques for solution space evolution are mainly applied for source code [ST00; ALR+05; AMC+07; LSK+13; NBA+15], whereas we focus on models, i.e., state machines, for test modeling. In the context of model-based SPL evolution [DGR+10; HRR+12; SSA13a; KLL+14; LNT+18], the transformational variability implementation technique delta modeling [Sch10; CHS15] is mostly used for different types of models, e.g., software architectures [HRR+12; KLL+14]. Delta modeling [Sch10; CHS15] is well-suited for handling variability and evolution for various types of artifacts as transformations are flexible enough to capture changes due to variation and evolution by the same means, i.e., deltas. However, none of those delta-oriented techniques [DGR+10; HRR+12; SSA13a; KLL+14] completely fulfill the four requirements, e.g., the documentation of the evolution history or the application of techniques for evolution analysis, to be ensured by a test-modeling formalism for evol-

ving SPLs as defined in the beginning of this chapter. Hence, an integrated test-modeling formalism for evolving SPLs is much needed facilitating efficient model-based regression testing (cf. Chapt. 5). Such a formalism has (1) to handle both, variability and evolution in the same way, (2) to be adaptable and, thus, applicable for various test-model artifact types, (3) to capture the complete evolution history, and (4) to facilitate the reasoning about the evolution impact.

3.3.1 Higher-Order Delta Modeling

We lift the notion of delta modeling [Sch10; CHS15] to define higher-order delta modeling as its extension to cope with evolving SPLs. For a more intuitive definition, we instantiate and describe higher-order delta modeling based on the delta state machine adaptation (cf. Sect. 3.2) in order to apply it as test-modeling formalism integrated in our regression testing framework (cf. Chapt. 5). However, similar to delta modeling [Sch10; CHS15], higher-order delta modeling is instantiable also for other types of domain artifacts, where a respective delta modeling adaptation exists for such as software architectures [HKR+11b; HKR+11a; HRR+12; LLL+14; KLL+14; LLL+15].

In higher-order delta modeling, we capture the evolution of the delta model of an SPL by means of higher-order deltas. A *higher-order delta* collects the evolution change operations to transform a delta model DM_θ of a version θ into its subsequent version $DM_{\theta'}$ of version θ' by altering the existing set of deltas Δ_{DM_θ} of DM_θ. Therefore, an *evolution change operation* specifies the addition, the removal, or the modification of a delta δ. A modification either alters the change operation set OP_δ of the delta δ to be modified by adding or removing change operations, exchanges the application condition φ_δ such that the delta is applicable for another set of variants, or exchanges the region r_ϵ to which the contained change operations are applied. We incorporate the modification of a delta as explicit evolution operations for a complete definition of higher-order delta modeling, yet an encoding of modifications via a removal of the original delta and an addition of the modified delta is also valid.

Definition 3.14: Evolution Change Operation

Let OP^H be the set of all evolution change operations defined over the universe $\mathcal{OP}$ of change operations. An *evolution change operation* $op^H \in OP^H$ defines one of the following transformations:

- add δ, i.e., a delta δ is added to a delta model,
- rem δ, i.e., a delta δ is removed from a delta model,
- mod $(\delta, \{\text{add } op, \text{rem } op', \ldots\})$, i.e., a delta δ is modified by altering the set of encapsulated change operation OP_δ,
- mod $(\delta, \varphi'_\delta)$, i.e., a delta δ is modified by exchanging the application condition φ_δ, or
- mod (δ, r'_ϵ) , i.e., a delta δ is modified by exchanging the region r_ϵ the change operations are applied to

For the addition of a new delta as well as the modification of the application condition of an existing delta, we require a valid feature model fm_{θ_i} for the version $\theta_i \in \Theta$ for which the evolution change operation has to be applied. The feature model fm_{θ_i} facilitates (1) the correct definition of application conditions φ_δ of a delta δ, and (2) the validation whether a delta δ of the delta model DM_{θ_i} is applicable for a variant-specific state machine $sm_v^{\theta_i}$ in version θ_i. We assume a feature model fm_{θ_i} for a version $\theta_i \in \Theta$ as given as we abstract from the evolution of feature models. For respective

techniques, we refer to the literature [BPD+10; PBD+12; SSA13a; NSS16]. To specify for which version $\theta_i \in \Theta$ a higher-order delta has to be applied to transform the preceding delta model version $DM_{\theta_{i-1}}$ into the new one DM_{θ_i}, we directly map a higher-order delta to its version θ_i.

Definition 3.15: Higher-Order Delta

A *higher-order delta* $\delta^H = (OP^H_{\delta^H}, \theta)$ is a tuple, where

- $OP^H_{\delta^H} = \{op^H_1, \ldots, op^H_l\} \subset OP^H$ is a finite set of *evolution change operations*, and
- $\theta \in \Theta$ is the *version* for which the higher-order delta has to be applied to transform the previous delta model version $DM_{\theta'}$ into the new one DM_θ.

In the remaining thesis, we use the shorter notation δ^H_θ for a version-specific higher-order delta. Each higher-order delta δ^H_θ represents an *evolution step* of the complete *evolution history* of an SPL, which we capture as a higher-order delta model. A *higher-order delta model* encapsulates the set of version-specific higher-order deltas $\delta^H_{\theta_i}$ as well as an initial delta model $DM_\emptyset$ which is transformed to obtain version-specific delta models DM_{θ_i} of versions $\theta_i \in \Theta$. The initial delta model $DM_\emptyset$ is defined as empty delta model comprising the core model $sm_{v_{core}}$ and an empty set of deltas $\Delta_{DM_\emptyset} = \emptyset$ to build the basis for the delta model evolution. Thus, the initial higher-order delta $\delta^H_{\theta_0}$ captures solely additions of deltas such that the initial empty delta model $DM_\emptyset$ is completed to DM_{θ_0} to comply to the initial SPL version θ_0. For higher-order delta modeling, we apply the concept of pure delta modeling as it facilitates an easier handling of SPL evolution [SD10]. Furthermore, by using an empty delta model as initial delta model $DM_\emptyset$, we allow for a consistent definition of a higher-order delta model, where every SPL version and its delta model is created based on the application of a higher-order delta. However, as pure and core delta modeling are equivalent modeling strategies [SD10], our technique also facilitates the definition of a higher-order delta model based on a non-empty initial delta model.

Definition 3.16: Higher-Order Delta Model

Let Δ^H be the set of all higher-order deltas definable based on the set OP^H of all evolution change operations. A *higher-order delta model* $DM^H_\Theta = (\Delta^H_\Theta, DM_\emptyset)$ of an SPL with versions Θ is defined as tuple, where

- $\Delta^H_\Theta = \{\delta^H_{\theta_0}, \ldots, \delta^H_{\theta_n}\} \subset \Delta^H$ is a finite set of version-specific *higher-order deltas*, and
- $DM_\emptyset = (sm_{v_{core}}, \emptyset)$ is the initial *empty delta model*.

By Δ^H, we refer to the set of all higher-order deltas definable based on the set OP^H of all evolution change operations. For the transformation of the initial delta model $DM_\emptyset$ into a delta model DM_{θ_i} of an SPL version $\theta_i \in \Theta$, the preceding higher-order deltas $\delta^H_{\theta_j} \in \Delta^H_\Theta$ are sequentially applied starting with the initial higher-order delta $\delta^H_{\theta_0}$. Therefore, the application sequence of higher-order deltas is predefined based on the strict total order of the SPL versions Θ such that

$$\mathsf{apply}_{\delta^H}(DM_\emptyset, (\delta^H_{\theta_0}, \ldots, \delta^H_{\theta_{i-1}}, \delta^H_{\theta_i})) = DM_{\theta_i}$$

holds. Similar to the state machine delta application (cf. Def. 3.12), the function $\mathsf{apply}_{\delta^H}$ defines the incremental application of higher-order deltas on an existing delta model.

Definition 3.17: Higher-Order Delta Application

The sequential application of higher-order deltas is defined via the function

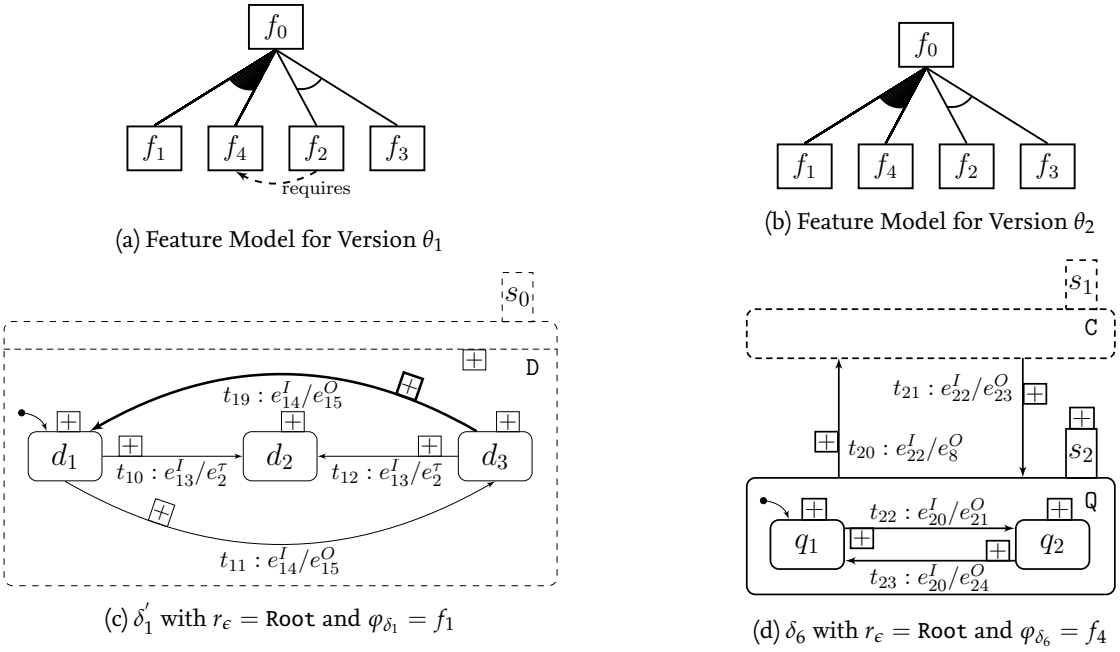
$$\mathsf{apply}_{\delta^H} : \mathcal{DM} \times \Delta^{H+} \to \mathcal{DM}$$

as follows:

- $\mathsf{apply}_{\delta^H}(DM_\emptyset, (\delta^H_{\theta_0}, \ldots, \delta^H_{\theta_{i-1}}, \delta^H_{\theta_i})) = \mathsf{apply}_{\delta^H}(\mathsf{apply}_{\delta^H}(DM_\emptyset, \delta^H_{\theta_0}), (\delta^H_{\theta_1}, \ldots, \delta^H_{\theta_{i-1}}, \delta^H_{\theta_i}))$
- $\mathsf{apply}_{\delta^H}(DM, \delta^H) = \mathsf{apply}_{\delta^H}(DM, \{op^H_1, \ldots, op^H_l\})$
- $\mathsf{apply}_{\delta^H}(DM, \{op^H_1, \ldots, op^H_l\}) = \mathsf{apply}_{\delta^H}(\mathsf{apply}_{\delta^H}(DM, op^H_1), \{op^H_2, \ldots, op^H_l\})$
- $\mathsf{apply}_{\delta^H}(DM, \mathsf{add}\,\delta) = DM' = (sm_{v_{core}}, \Delta_{DM} \cup \{\delta\})$
- $\mathsf{apply}_{\delta^H}(DM, \mathsf{rem}\,\delta) = DM' = (sm_{v_{core}}, \Delta_{DM} \setminus \{\delta\})$
- $\mathsf{apply}_{\delta^H}(DM, \mathsf{mod}\,(\delta, \{\mathsf{add}\,op, \mathsf{rem}\,op', \ldots\})) =$
  $DM' = (sm_{v_{core}}, (\Delta_{DM} \setminus \{\delta\}) \cup \{\delta'\})$ with $\delta = (\{op_1, \ldots, op', \ldots, op_k\}, r_\epsilon, \varphi_\delta)$
  and $\delta' = (\{op_1, \ldots, op, \ldots, op_k\}, r_\epsilon, \varphi_\delta)$
- $\mathsf{apply}_{\delta^H}(DM, \mathsf{mod}\,(\delta, \varphi'_\delta)) = DM' = (sm_{v_{core}}, (\Delta_{DM} \setminus \{\delta\}) \cup \{\delta'\})$
  with $\delta = (\{op_1, \ldots, op_k\}, r_\epsilon, \varphi_\delta)$ and $\delta' = (\{op_1, \ldots, op_k\}, r_\epsilon, \varphi'_\delta)$
- $\mathsf{apply}_{\delta^H}(DM, \mathsf{mod}\,(\delta, r'_\epsilon)) = DM' = (sm_{v_{core}}, (\Delta_{DM} \setminus \{\delta\}) \cup \{\delta'\})$
  with $\delta = (\{op_1, \ldots, op_k\}, r_\epsilon, \varphi_\delta)$ and $\delta' = (\{op_1, \ldots, op_k\}, r'_\epsilon, \varphi'_\delta)$

A higher-order delta model DM^H_Θ captures and, therefore, documents the evolution history $\Theta = \{\theta_0, ..., \theta_n\}$ up to the present version θ_n. However, evolution is not a limited process such that new evolution steps emerge in the future, e.g., based on upcoming customer requests [SB99; SV02; MSC14; BP14]. To integrate a new evolution step for version θ_{n+1}, we first define a new higher-order delta $\delta^H_{\theta_{n+1}}$ comprising the evolution change operations, e.g., the addition of a delta or the modification of an application condition, that transform the last delta model DM_{θ_n} of version θ_n into the delta model $DM_{\theta_{n+1}}$ of version θ_{n+1}, and map it to the new version θ_{n+1}. Second, we integrate the new higher-order delta $\delta^H_{\theta_{n+1}}$ into the existing higher-order delta model $DM^H_\Theta = (\Delta^H_\Theta, DM_\emptyset)$ such that $\Delta^H_\Theta = \{\delta^H_{\theta_0}, \ldots, \delta^H_{\theta_n}, \delta^H_{\theta_{n+1}}\}$ holds. To ensure consistency between problem as well as solution space evolution, we also integrate a new higher-order delta if solely the feature model evolves to correspond to a new SPL version. In such a case, e.g., the feature model is refactored for upcoming evolution steps, the set of variants and, therefore, the domain artifacts remain stable. Thus, the new higher-order delta captures an empty set of evolution change operations to represent the fact that no changes on the solution space level are to be made.

Furthermore, higher-order delta modeling facilitates a straightforward way to adapt the core $sm_{v_{core}}$ during the evolution of an SPL. By adding a new delta $\delta_{sm_{v_{core}}}$ via a higher-order delta δ^H, we define the respective application condition $\varphi_{\delta_{sm_{v_{core}}}} = \texttt{true}$ such that the new delta is always applied to transform the core independent from a concrete feature configuration. This procedure weakens the concerns of Schaefer and Damiani [SD10] that core delta modeling [Sch10; SBB+10; CHS15] does not facilitate an intuitive evolution of the core, but we still have to maintain the core-changing delta $\delta_{sm_{v_{core}}}$ in upcoming evolution steps instead of directly adapting the core. However, higher-order delta modeling is also applicable to pure delta modeling [SD10; SBB+10; CHS15], where we are able to modify the deltas directly which are responsible to create the core.



(a) Feature Model for Version θ_1

(b) Feature Model for Version θ_2

(c) δ_1' with $r_\epsilon = \texttt{Root}$ and $\varphi_{\delta_1} = f_1$

(d) δ_6 with $r_\epsilon = \texttt{Root}$ and $\varphi_{\delta_6} = f_4$

Figure 3.6: Sample Evolution of Feature Model and Delta Model

Example 3.5: Higher-Order Delta Modeling

Consider the feature model fm_{θ_0} from Ex. 2.1 shown in Fig. 2.5 defined for the initial version θ_0 of our running SPL example. The delta model $DM_{\theta_0} = (sm_{v_{core}}, \{\delta_1, \ldots, \delta_5\})$ for version θ_0 is specified in Ex. 3.3, where the core state machine is depicted in Fig. 3.1 and the five deltas are shown in Fig. 3.3. To obtain this delta model, we define the initial higher-order delta $\delta^H_{\theta_0} = (\{\mathsf{add}\,\delta_1, \mathsf{add}\,\delta_2, \mathsf{add}\,\delta_3, \mathsf{add}\,\delta_4, \mathsf{add}\,\delta_5\}, \theta_0)$ such that all five deltas are added. The higher-order delta model $DM^H_\Theta = (\{\delta^H_{\theta_0}\}, DM_\emptyset)$ captures the empty delta model $DM_\emptyset$ and the initial higher-order delta for the initial version θ_0 of the running example. In Fig. 3.6a, the evolved feature model fm_{θ_1} of SPL version θ_1 is depicted. We added the feature f_4 and combined the new feature with the feature f_1 in an or-feature-group. We also added the constraint that f_2 requires the selection of f_4. The feature model changes result in a change of the variant-specific feature configurations and, hence, of the variant set. The feature configuration $F_{v_{core}}$ of the core v_{core} is modified to $F_{v'_{core}} = \{f_0, f_2, f_4\}$, F_{v_1} of variant v_1 is modified to $F_{v'_1} = \{f_0, f_2, f_1, f_4\}$, F_{v_2} is modified to $F_{v'_2} = \{f_0, f_3, f_4\}$, $F_{v_3} = \{f_0, f_3, f_1\}$ is unchanged, and $F_{v_4} = \{f_0, f_3, f_1, f_4\}$ is new. To evolve the delta model DM_{θ_0} to its next version DM_{θ_1}, the higher-order delta $\delta^H_{\theta_1} = (\{\mathsf{mod}\,(\delta_1, \{\mathsf{add}\,t_{19}\}), \mathsf{add}\,\delta_6\}, \theta_1)$ is defined. The modified delta δ_1' is depicted in Fig. 3.6c, where the added operation is highlighted. In addition, the new delta δ_6 is shown in Fig. 3.6d. To document the evolution history, the higher-order delta model $DM^H_\Theta = (\{\delta^H_{\theta_1}, \delta^H_{\theta_1}\}, DM_\emptyset)$ is extended accordingly. For SPL version θ_2, assume that solely changes on the feature model level are required. For instance, we remove the requires edge to allow for a generalization of the variant set [TBK09] as we now are able to derive $F_{v_5} = \{f_0, f_2, f_1\}$ as new variant v_5. Accordingly, we define an empty higher-order delta $\delta^H_{\theta_2} = (\emptyset, \theta_2)$ and integrate it in DM^H_Θ for the complete evolution documentation.

Similar as for delta modeling [Sch10; CHS15], we obtain a higher-order delta model by creating it (1) manually with a corresponding tool support or (2) automatically using model differencing techniques [PKK+15; PRK+17]. We describe our prototypical implementation in Sect. 3.4. Again, for our testing framework (cf. Chapt. 5), we abstract from the concrete creation, but assume a higher-order delta model as given representing the behavioral specification of an evolving SPL under test.

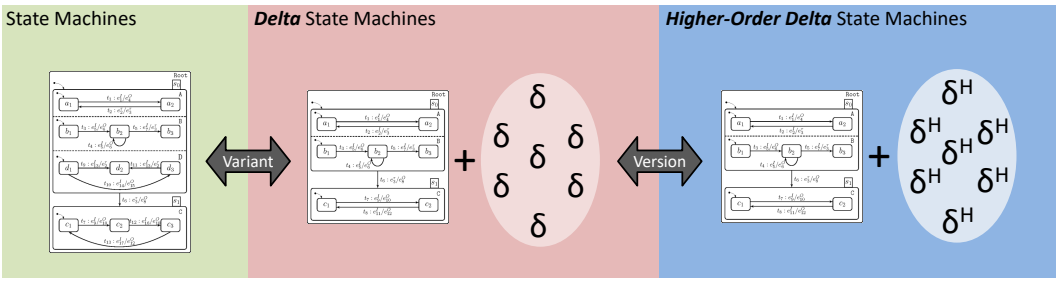


Figure 3.7: Overview and Relation of Delta-Oriented Test Modeling for Variants and Versions of Variants

The relation between test modeling for a single variant by means of state machines, for an SPL version in terms of delta state machines, and for the complete evolution history via higher-order delta state machines is depicted in Fig. 3.7. Furthermore, we summarize the list of symbols used for the definition of higher-order delta modeling in Tab. 3.3. To recapitulate, a higher-order delta $\delta^H = (OP^H_{\delta^H}, \theta)$ encapsulates a set $OP^H_{\delta^H}$ of evolution change operations, e.g., the addition of a state machine delta, and is mapped to a specific SPL version $\theta \in \Theta$ of the complete SPL evolution history. The complete evolution of an SPL is captured by a higher-order delta model $DM^H_\Theta = (\Delta^H_\Theta, DM_\emptyset)$ combining a set Δ^H_Θ of higher-order deltas and the initial empty delta model $DM_\emptyset$ solely containing a state machine used as core for the evolving SPL. To obtain the delta model DM_{θ_i} of a specific version θ_i of the evolution history, we incrementally apply the sequence $(\delta^H_{\theta_0}, \ldots, \delta^H_{\theta_{i-1}}, \delta^H_{\theta_i})$ of higher-order deltas to the initial delta model $DM_\emptyset$ using the application function $\mathsf{apply}_{\delta^H}$.

Table 3.3: Symbol Summary of Higher-Order Delta Modeling Definition

Symbol	Description
$\theta; \Theta$	SPL version; SPL evolution history
$op^H; OP^H$	Evolution change operation; Set of evolution change operations
$\delta^H; \Delta^H_\Theta$	Higher-order delta; Set of higher-order deltas
DM^H_Θ	Higher-order delta model
$\mathsf{apply}_{\delta^H}$	Higher-order delta application function

3.3.2 Benefits and Limitations

Higher-order delta modeling has benefits and limitations for its application as a formalism for test modeling supporting the test process of evolving SPLs, in particular, and its application in SPLE [PBvdL05], in general, discussed in the following.

Benefits

The main benefits of higher-order delta modeling are given by addressing the four requirements defined in the beginning of this chapter, namely (1) handling variability and evolution in an inte-

grated way, (2) being adoptable for different artifact types, (3) documenting of the evolution history, and (4) supporting the application of the analysis of evolution aspects. Those requirements are crucial for a test-modeling formalism to allow for efficient testing of evolving SPLs as follows.

Integrated Modeling. Higher-order delta modeling represents an integrated variability modeling technique as we handle variant and version information as first-class entities. We lift and, thereby, extend the concept of delta modeling [Sch10; CHS15] to capture the variability and the evolution of an SPL by the same means, i.e., (higher-order) deltas. The consistent modeling improves the comprehensibility as well as the modelability of evolving SPLs. Hence, the application of higher-order delta modeling as test modeling, in particular, or as implementation technique for SPLE [PBvdL05], in general, supports a test engineer/developer facilitating the development of evolving SPLs. Again, similar as for delta modeling [Sch10; CHS15], we are able to obtain a higher-order delta model by creating it (1) manually with a corresponding tool support or (2) automatically using, e.g., model differencing techniques [PKK+15; PRK+17]. This way, our modeling technique allows for the extractive, reactive, and proactive development [Kru02] of evolving SPLs. In this context, the respective tool support is an important aspect. Furthermore, depending on the application scenario, higher-order delta modeling is not solely useful as variability implementation technique [SRC+12], but is also exploitable as some kind of data structure in variation control systems [LBG17].

Adaptability. As higher-order delta modeling extends delta modeling [Sch10; CHS15], we inherit its adaptability to various artifact types. This fact enables a wide range of possible applications. In this thesis, we instantiated (higer-order) delta modeling for state machines which are a well-established modeling formalism already employed in the context of model-driven development [Har87; HP98; Obj09; Col06] and quality assurance, e.g., model-based component testing [UL06; Wei10; LPK+14; Loc13; LTW+14]. In addition, we are able to instantiate our formalism for other artifact types such as Matlab/Simulink [HKM+13] for the development in the automotive domain [Hol12], for source code [SBB+10; SD10; KHS+14] to implement evolving SPLs, or software architectures [HKR+11a; HRR+12; LLL+14; LLL+15] to facilitate model-based incremental integration testing.

Evolution History. Another aspect to be taken into account when managing the evolution of SPLs is the respective documentation of the evolution history. Design decisions made in preceding evolution steps may introduce inconsistencies in combination with changes which are to be made to step to the next SPL version. Similar as for feature models [NST18; NMS+18], we are able to determine the evolution step in which the respective changes were applied resulting in an improved understanding of the evolution of an SPL under consideration. Furthermore, the documentation of the evolution history supports the planning of anticipated evolution [SPB+12] in the future by incorporating previous evolution steps and their impact on the SPL.

Analysis of Evolution. As the last benefit, higher-order delta modeling facilitates the application of change impact analysis (cf. Chap. 4). Change impact analysis is a crucial factor for efficient and effective regression testing [YH12] in order to reduce the testing redundancy by focusing on changes and their impact caused by an evolution step. Higher-order deltas specify how the delta set of a version-specific delta model changes in terms of additions, removals, and modifications of deltas. We are able to take those changes into account and directly pass on them to the variant-specific delta sets in order to infer and reason about the impact of the application of a higher-order delta to the respective variants. For the reasoning process, we exploit the commutativity of higher-order

delta application and delta-oriented variant generation [LKS16]. We refer to Lity et al. [LKS16] for the proof of the commutation based on a general artifact-independent definition of higher-order delta modeling. Hence, we are able to identify how the variant set changes between subsequent SPL versions under test in terms of additions, removals, and modifications of variants as described in Sect. 4.2. Furthermore, higher-order delta modeling facilitates the analysis of particular variants and their versions in order to identify the impact of changes between them. In general, our modeling formalism supports incremental analysis techniques [TAK+14] exploiting the explicit knowledge about differences which, in our case, are captured as (higher-order) deltas or regression deltas.

Limitations

Besides the described benefits, higher-order delta modeling also has some limitations. The main limitation is the size and, therefore, complexity of a higher-order delta model which increases with every new evolution step of an evolving SPL. In general, this is a drawback of variability implementation techniques as variability introduces a new dimension of complexity which is even more the case when incorporating evolution as second dimension for the development of SPLs. That means, a user of higher-order delta modeling has to know (1) how the version-specific delta models are composed by deltas, and (2) how a version-specific delta model can be transformed by altering its delta set. To alleviate this drawback, a user has to be supported by an adequate tool support, where, for instance, based on respective views, the modeling complexity is reduced.

Another limitation is the restricted potential of the efficient application of different analysis strategies [TAK+14]. Higher-order delta modeling facilitates the support for efficient incremental analysis such as change impact analysis (cf. Sect. 4.2). For the application of further analysis strategies [TAK+14], e.g., family-based analysis, respective techniques have to be adapted or newly realized. For instance, Damiani et al. [DS12] proposed family-based type checking for delta-oriented SPLs which could be extended to also analyze evolving delta-oriented SPLs which are captured based on higher-order delta modeling. Another solution to cope with this restriction are transformations between variability implementation techniques also called variant-preserving mapping according to Fenske et al. [FTS13] to apply different analysis strategies [LNT+18; LRB+19].

3.4 Tool Support and Sample Application

In this section, we present our prototypical implementation supporting delta-oriented test modeling of variants and versions of variants. Furthermore, we introduce three evolving delta-oriented product lines to be applied as subject systems for the evaluation of our change impact analysis techniques in Chapt. 4 as well as our model-based regression testing framework in Chapt. 5.

3.4.1 Prototype

We provide a prototypical tool support called DOPE (**D**elta-**O**riented **P**roduct-Line **E**volution) for our delta-oriented test modeling of variants and versions of variants realized as ECLIPSE[1] plug-ins. The plug-ins are created based on the ECLIPSE Modeling Framework[2] (EMF). EMF facilitates the model-driven development of ECLIPSE plug-ins by specifying meta models from which respective JAVA source code and editors for viewing and modeling purposes are automatically generated. In parti-

[1] https://www.eclipse.org/, last access: May 31st, 2019
[2] https://www.eclipse.org/modeling/emf/, last access: May 31st, 2019

cular, the following plug-ins are specified for delta-oriented test modeling based on corresponding meta models:

- `de.imotep.variability.featuremodel` – Plug-in to allow for feature modeling (cf. Sect. 2.2.2).
- `de.imotep.variability.configuration` – Plug-in to allow for the management of feature configurations (cf. Sect. 2.2.2).
- `de.imotep.core.behavior` – Plug-in to allow for state machine modeling (cf. Sect. 3.1).
- `de.imotep.variability.deltaBehavior` – Plug-in to allow for delta state machine modeling (cf. Sect. 3.2).
- `de.imotep.dope` – Plug-in to allow for higher-order delta state machine modeling (cf. Sect. 3.3).

The plug-ins are part of the tool support of the research project IMoTEP[3] and, hence, developed in cooperation with the Real-Time Systems Lab[4] of the Technische Universität Darmstadt. The IMoTEP project focuses on efficient and effective model-based testing of evolving (dynamic) SPLs. Based on the usage of EMF and the partition of the plug-in functionalities, we are able to improve and extend our prototype in the future, e.g., to support incremental SPL integration testing [LLL+14; LLL+15; LLA+16] using software architectures as delta-oriented test models. In the following paragraphs, we show and describe the meta models of the plug-ins, where we abstract from those parts of the meta models which are not required for delta-oriented state machine test modeling.

Feature Modeling. We require the feature model plug-in `de.imotep.variability.featuremodel` to allow for (1) the definition of valid feature configurations, and (2) the definition and evaluation of application conditions of state machine deltas. The main class of the meta model shown in Fig. 3.8 is `FeatureModel`. A `FeatureModel` has as starting point one `Feature` as `root` feature. The further hierarchical decomposition is handled via `FeatureGroup`. Each `Feature` which is not the `root` feature of a `FeatureModel` is either contained in `AlternativeFeatureGroup`, `OrFeatureGroup`, or `AndFeatureGroup`. In case `Feature` is contained in a `AndFeatureGroup`, its `variabilityType` attribute defines whether `Feature` is `MANDATORY` or `OPTIONAL`. In addition to the set of `Features`, a `FeatureModel` may comprise a set of `Constraints`. The class `Constraint` enables the definition of cross-tree constraints, i.e., require and exclude constraints specified in propositional logic in the attribute `code`. Based on the class `FeatureModelTransformer` (not shown in Fig. 3.8), we further incorporate the external tool FeatureIDE [MTS+17] in our prototype such that a feature model is modeled in FeatureIDE and imported as instance of our meta model via the method `importModel()`. In general, our feature modeling plug-in is independent from tools such as FeatureIDE. Hence, we are able to replace FeatureIDE by alternative tools with similar functionality.

Feature Configuration Management. The plug-in `de.imotep.variability.configuration` allows for the definition and management of feature configurations required to determine the set of applicable state machine deltas for a respective variant. The main class of the meta model depicted in Fig. 3.9 is `FeatureConfigurationManager`. The class comprises the set of `FeatureConfigs` representing feature configurations $F_v \in F_{\mathbb{V}}$. Each `FeatureConfig` references its set of `selected Features` and its set of `unselected Features`. To facilitate a correct referencing, the class `FeatureConfigurationManager` is mapped to a `FeatureModel` containing the referenced `Feature`. Based on the method `convertToAnnotationString()`, the feature configuration is pretty printed in con-

[3] http://www.dfg-spp1593.de/imotep/, last access: May 31st, 2019
[4] https://www.es.tu-darmstadt.de/en/es-real-time-systems-lab/, last access: May 31st, 2019

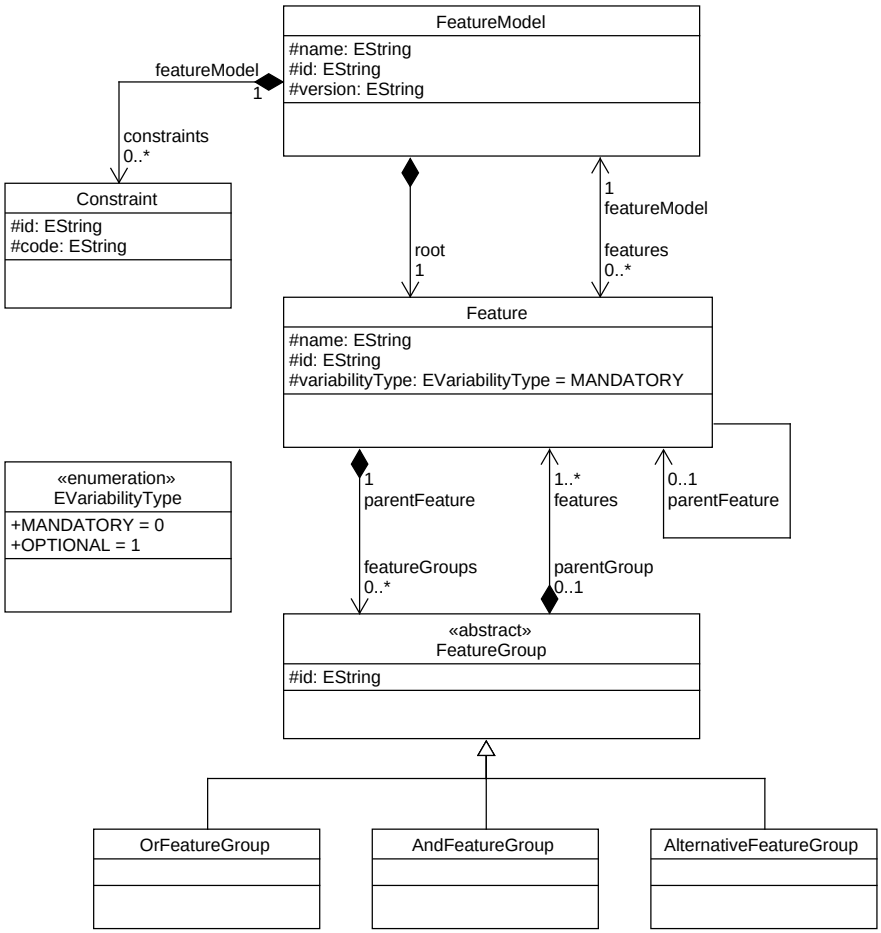


Figure 3.8: Meta Model of the Feature Model Plug-In

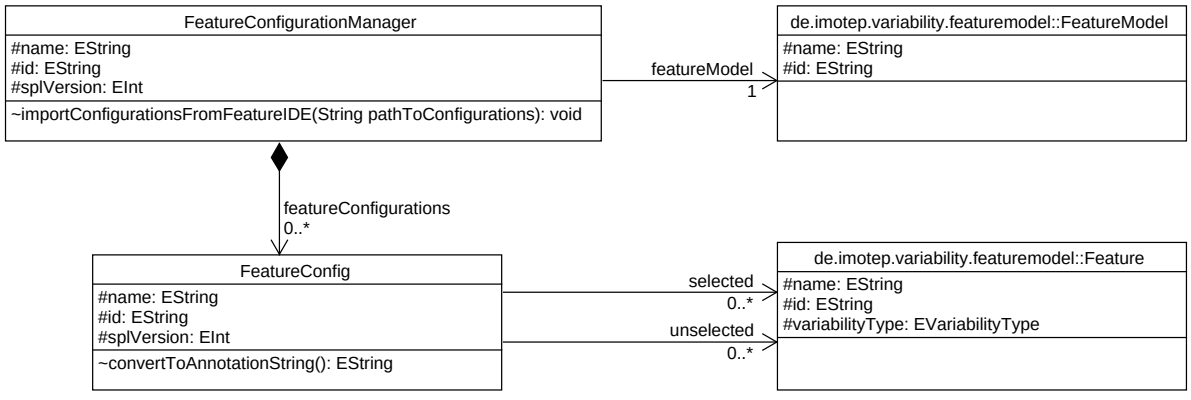


Figure 3.9: Meta Model for Feature Configurations Management Plug-In

junctive normal form. Similar to the import of a feature model from FeatureIDE [MTS+17], we also allow for the import of the variant-specific feature configurations F_{v_i} which are computed by FeatureIDE. Again, this is an additional functionality which can be replaced by another importer to support alternative feature modeling tools.

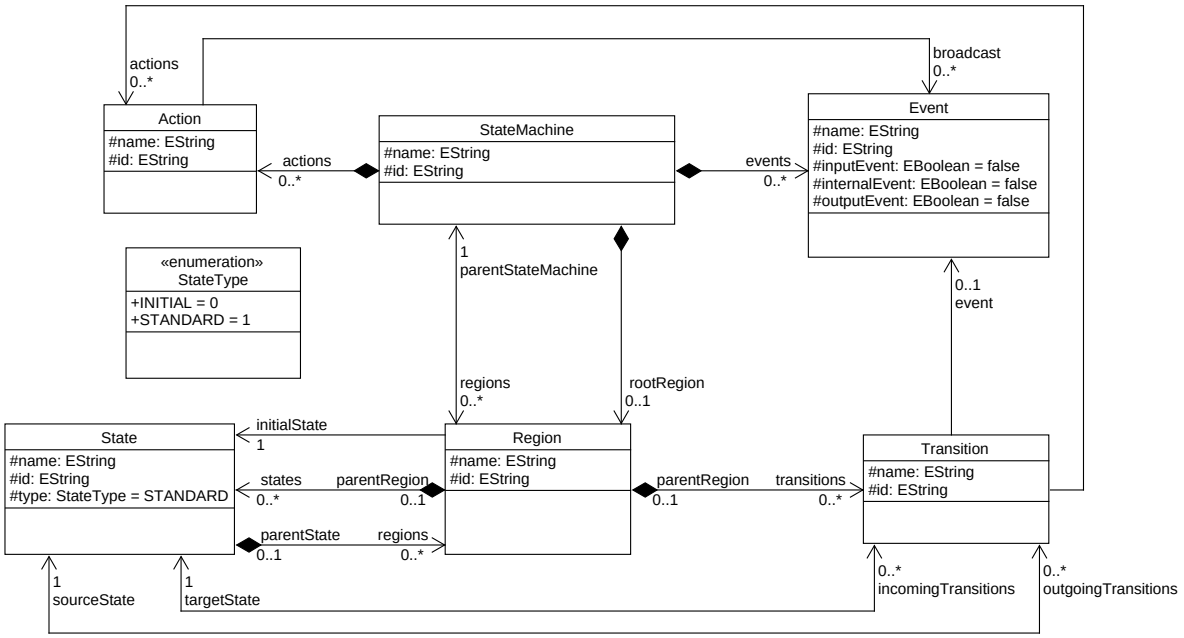


Figure 3.10: Meta Model for State Machine Modeling Plug-In

State Machine Modeling. The plug-in `de.imotep.core.behavior` builds the foundation for delta as well as higher-order delta modeling in our prototype. The main class of the meta model shown in Fig. 3.10 is `StateMachine`. As defined by the abstract syntax described above (cf. Sect. 3.1), a `StateMachine` comprises the sets of `Regions`, of `Events`, and of `Actions`. In addition, the designated `rootRegion` is referenced. Each `Region` contains the set of `States` and `Transitions` for specifying the behavior of the respective part of a system the `Region` represents. A `State` is either an `INITIAL` state or a `STANDARD` state denoted by the enumeration `StateType`. For each `State`, the incoming as well as outgoing `Transitions` are referenced and, furthermore, a `State` can again be hierarchically decomposed by a set of `Regions`. A `Transition` connects two `States` and optionally has an `Event` as trigger `event` as well as a set of `Actions` to define the `broadcast` of `Events`. Hence, we decompose a transition label by referencing to the respective EMF classes. All state machine elements, i.e., `Regions`, `States`, and `Transitions`, inherit from the superclass `BehaviorEntity`. The inheritance is used in the plug-in `de.imotep.variability.deltaBehavior` to facilitate a more general definition to which state machine element a change operation of a state machine delta is to be applied, e.g., the addition of a `State` is applied to a `Region`.

Delta State Machine Modeling. The plug-in `de.imotep.variability.deltaBehavior` adapts delta modeling [Sch10; CHS15] for state machines as described in Sect. 3.2. For a better illustration, we split the meta model in four parts. In Fig. 3.11, the main part of the meta model is shown,

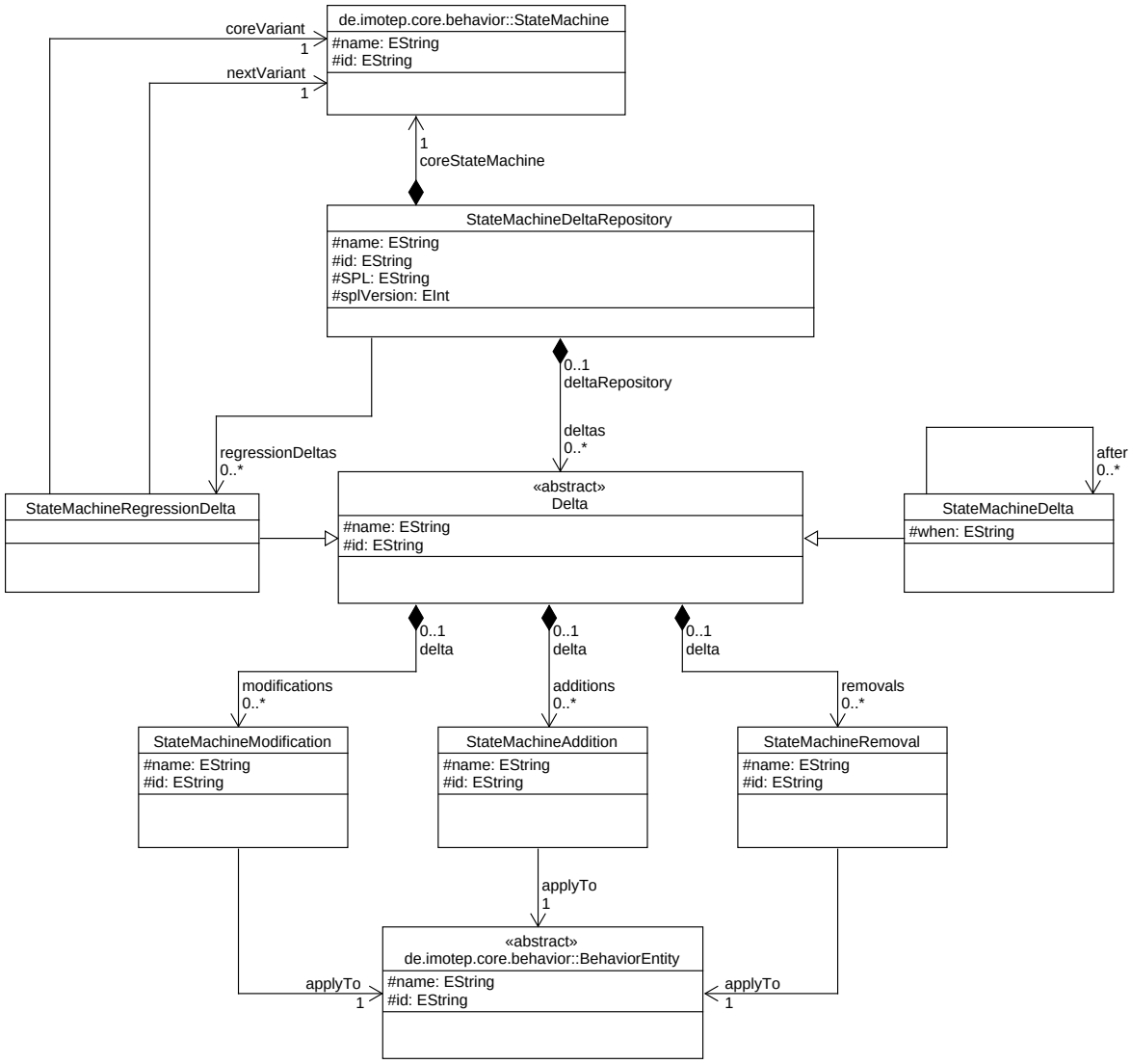


Figure 3.11: Meta Model for Delta State Machine Modeling Plug-In (Main Part)

whereas in Fig. 3.12 the addition part, in Fig. 3.13 the removal part, and in Fig. 3.14 the modification part are depicted. The main class of the meta model is `StateMachineDeltaRepository` representing a state machine delta model as defined above, where the `coreStateMachine` is referenced as well as the set of `deltas`. We use `Delta` as superclass to capture both `StateMachineDeltas` and `StateMachineRegressionDeltas` in a `StateMachineDeltaRepository`. A `Delta` comprises the set of change operations, i.e., `StateMachineModification` for modifications of transition labels, `StateMachineAddition` for additions, and `StateMachineRemoval` for removals of state machine elements. Each change operation is mapped to the class `BehaviorEntity` of `de.imotep.core.behavior` to directly specify to which state machine element the operation is to be applied. In contrast to the abstract syntax of our state machine dialect (cf. Sect. 3.1), where we reference the region r_e to which the delta is applied, we facilitate in the plug-in a more general definition in order to incorporate other state machine dialects and, therefore, other change operations in the future. Moreover, as

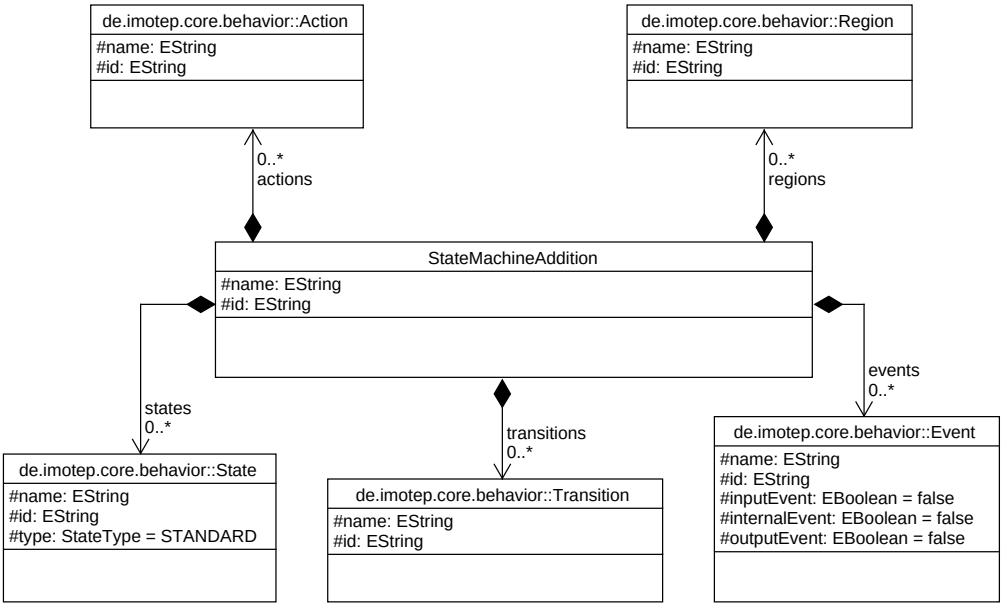


Figure 3.12: Meta Model for Delta State Machine Modeling Plug-In (Addition Part)

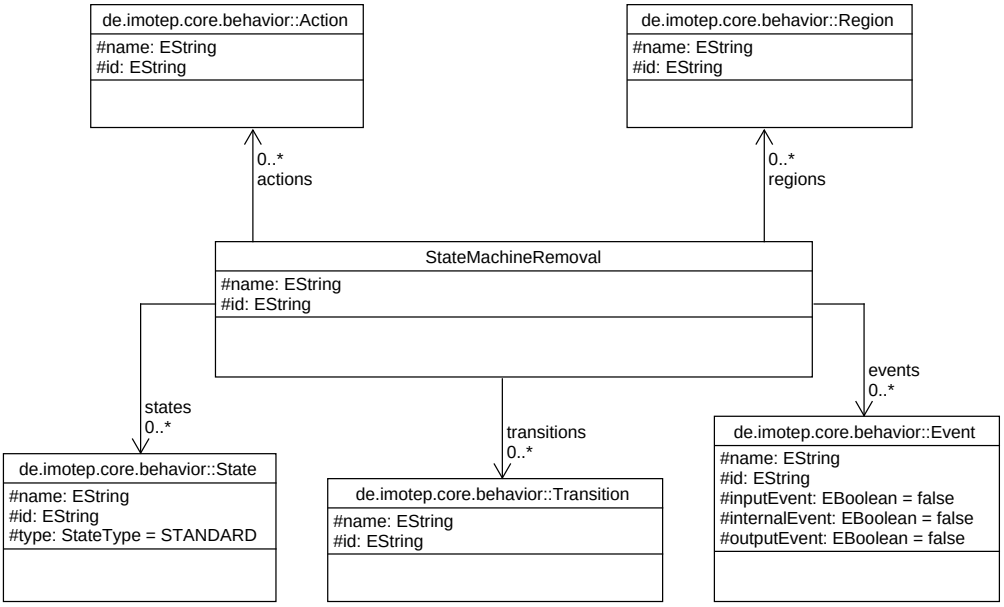


Figure 3.13: Meta Model for Delta State Machine Modeling Plug-In (Removal Part)

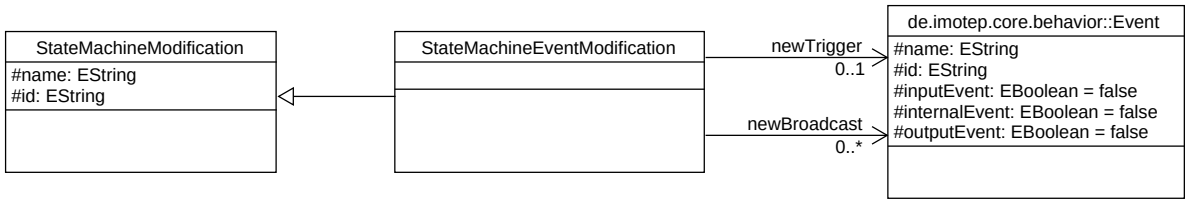


Figure 3.14: Meta Model for Delta State Machine Modeling Plug-In (Modification Part)

shown in Fig. 3.12 for `StateMachineAddition` and in Fig. 3.13 for `StateMachineRemoval`, we realize separate operations for `Actions` and `Events` due to the decomposition of a transition label in respective EMF classes. The modification of a transition label is represented by `StateMachineModification` setting `newTrigger` or `newBroadcast` as depicted in Fig. 3.14.

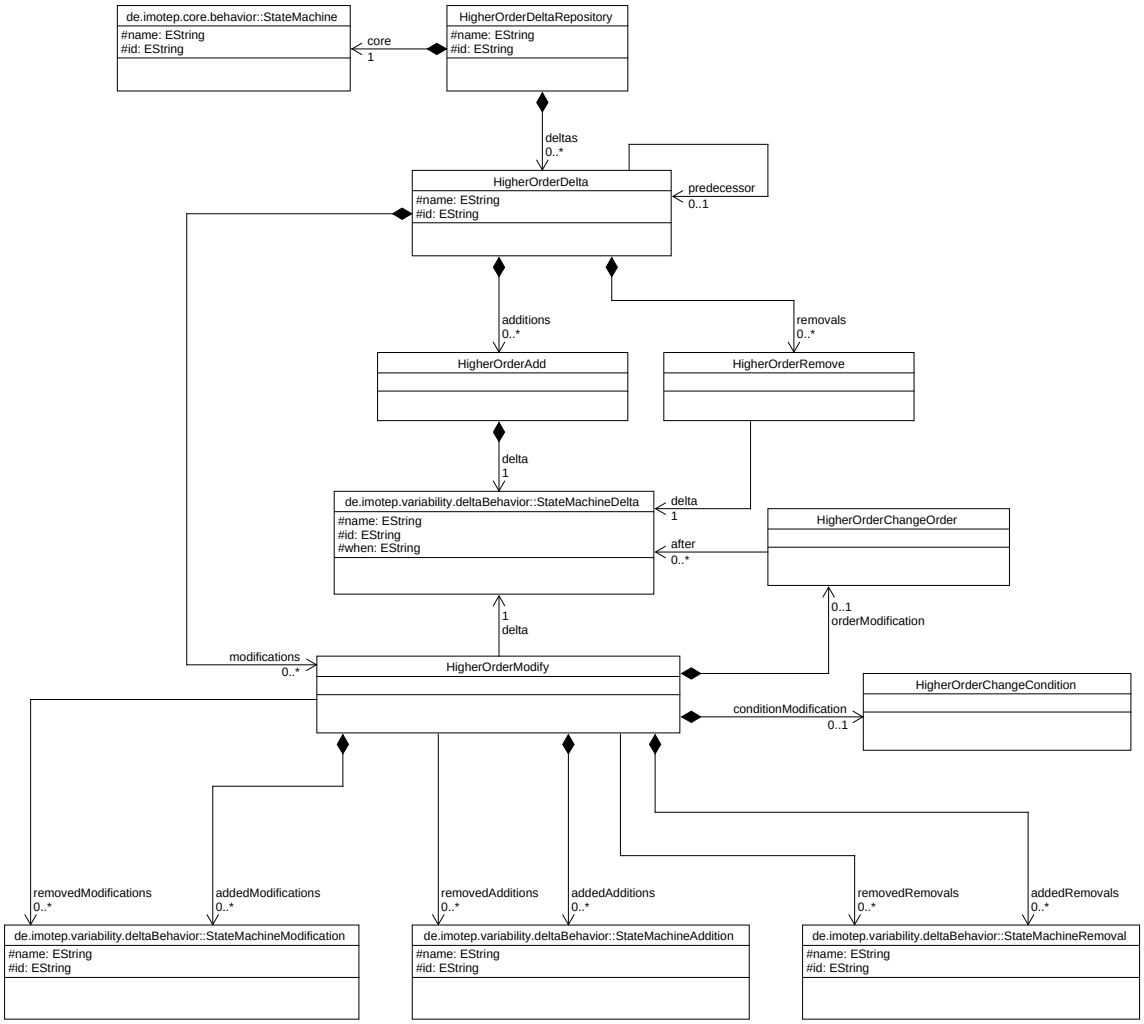


Figure 3.15: Meta Model for Higher-Order Delta State Machine Modeling Plug-In

Higher-Order Delta State Machine Modeling. The plug-in `de.imotep.dope` instantiates higher-order delta modeling (cf. Sect. 3.3) for delta state machines. As shown in Fig. 3.15, the main class of the meta model is `HigherOrderDeltaRepository` denoting a higher-order delta state machine model. Just as defined above, a `HigherOrderDeltaRepository` comprises a `StateMachine` as `core` and a set of `HigherOrderDeltas` as `deltas`. The order in which `HigherOrderDeltas` have to be applied is given based on the `predecessor` relation. Instead of a direct mapping to a version θ, we reference the `HigherOrderDelta` which has to be applied in advance. Each `HigherOrderDelta` comprises a set of evolution change operations implemented by the classes `HigherOrderAdd`, `HigherOrderRemove`, and `HigherOrderModify`. The classes `HigherOrderAdd` and `HigherOrderRemove` directly define the `StateMachineDelta` to be added or removed, respectively. In contrast, the class `HigherOrderModify` captures a `StateMachineDelta` to be modified and further the change operations to be added and removed, i.e., `StateMachineModification`, `StateMachineAddition`, and `StateMachineRemoval`. The class `HigherOrderModify` further defines how the application condition of a `StateMachineDelta` is altered via `HigherOrderChangeCondition` or how the set of `StateMachineDeltas` which have to be applied in advance can be altered via `HigherOrderChangeOrder`.

Based on those plug-ins, we are able to capture the evolution of model-based, i.e., state-machine-based SPLs. The prototype is provided online `https://github.com/SLity/mbtSPLregression`.

3.4.2 Evolving Subject Product Lines

The three evolving model-based software systems to be applied as subjects for the evaluation of our change impact analysis techniques and our model-based regression testing framework implement (1) a *Wiper SPL*, (2) a *Vending Machine SPL*, and (3) a *Mine Pump SPL*. Their original versions [Cla10] served already as benchmarks in the literature in the context of SPL quality assurance, e.g., for family-based verification [CCS+13] and incremental testing [LMT+16; LNT+19]. Those systems are suitable to be used in our evaluations as their event-based communication affects the application of our incremental change impact analysis (cf. Chapt. 4) and, hence, of our framework for model-based SPL regression testing (cf. Chapt. 5) guided by the impact analysis. Please note, although we focus on event-based systems and abstract from the read-/write-access of variables for the definitions of our contributions, those systems still comprise variables in order to provide also a synchronization between concurrent regions via shared variables. This fact further allows for an improved evaluation of our testing framework as we can investigate the effect of the neglection of shared variables on our retest test selection technique. For the evolution histories, we examined the original versions of the three subject systems and identified potential evolution scenarios to obtain distinct versions of each SPL [NLS18]. Hence, for each subject SPL, the evolution scenarios are based on each other and constitute a sequential evolution history captured via higher-order delta modeling and its prototypical tool support. The identified scenarios and their characteristics are also suitable for our evaluation as they affect our variant-set change impact analysis (cf. Sect. 4.2) in terms of removed, added, modified, and unchanged variants guiding the process of regression testing of subsequent SPL versions under test. We describe the three model-based SPLs and their versions shortly in the following, where we provide a small excerpt of the core state machine, initial delta set, and higher-order deltas of the Wiper SPL to show the sample application of higher-order delta modeling to the subject systems. For the complete documentation of the evolution history, we refer to the respective technical report [NLS18].

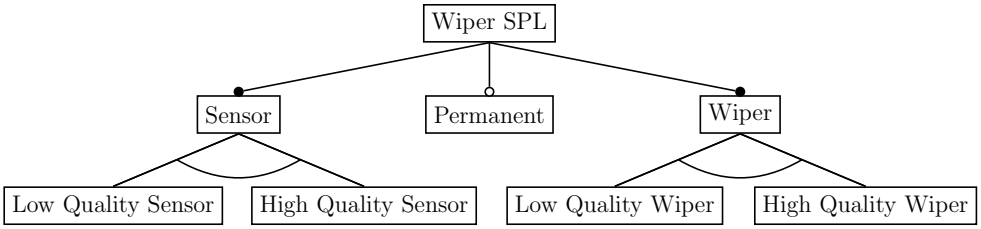


Figure 3.16: Feature Model of the Initial Wiper SPL Version (according to Classen [Cla10])

Wiper (W). The SPL specifies the variable control software of a car wiper system. Its original version was introduced by Gruler et al. [GLS08] and adapted by Classen [Cla10] using an annotative variability implementation technique [SRC+12] called featured transition systems [CCS+13]. The product line allows for the derivation of eight variants comprising variable qualities of rain sensors and wipers. In addition to the automatic wiping controlled either based on a high quality or low quality wiper and guided by either a high quality or low quality rain sensor, there is an optional permanent wiping mode. The corresponding feature model (taken from Classen [Cla10]) is shown in Fig. 3.16.

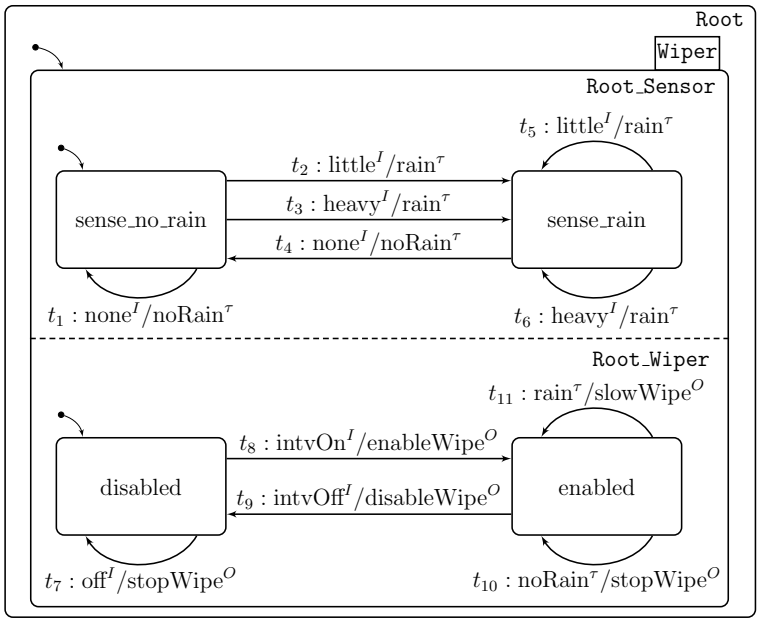
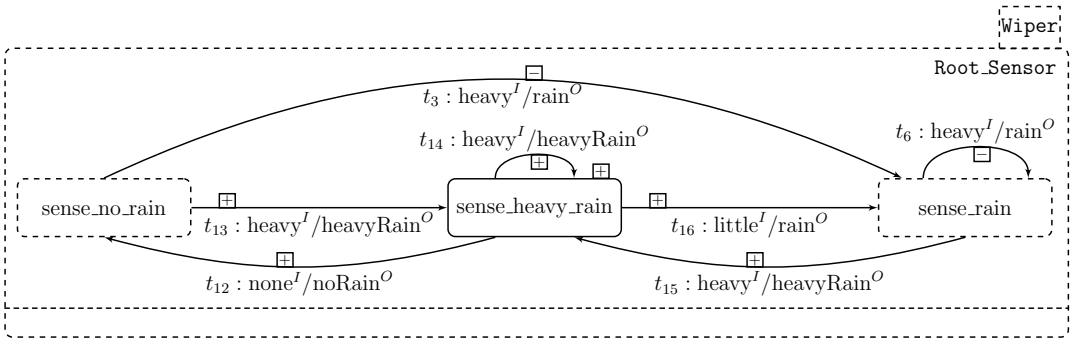


Figure 3.17: Core State Machine of the Initial Wiper SPL Version

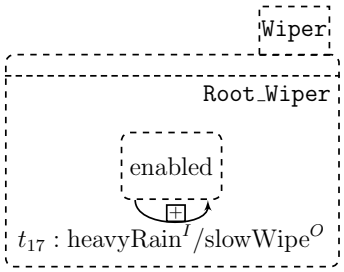
We remodeled the behavioral specification given as featured transition system to be represented by delta state machines [LMT+16; NLS18]. The core state machine of the delta-oriented system version is depicted in Fig. 3.17. The root region `Root` comprises the main computational state `Wiper` which is further decomposed in the concurrent regions `Root_Sensor` and `Root_Wiper`. The region `Root_Sensor` specifies the behavior of the rain sensors based on the two states `sense_no_rain` and `sense_rain` as well as six transitions. The transitions react on distinct rain intensities encoded as events with corresponding events for controlling the wipers of the core variant. Region `Root_Wiper` defines the behavior of the wiper actuators based on the two states `disabled` and `enabled` as well as five transitions. The transitions allow for switching on the wiper and for the reaction to the detected rain with corresponding wiping intensities. In Fig. 3.18, five deltas are shown used to

obtain the remaining variant-specific state machines of the Wiper SPL. For instance, the delta δ_{HS} depicted in Fig. 3.18a adds the state `sense_heavy_rain` as well as five transitions, and removes two transitions to facilitate the system to detect rain on a more fine-granular level, i.e., the sensor distinguishes between `little` and `heavy` rain. The deltas δ_{HSLW}, δ_{HSHW}, and δ_{LSHW} transform the core state machine depending on the combination of high as well as low level wipers and sensors. In addition, the delta δ_{Perm} shown in Fig. 3.18e adds the functionality of permanent wiping to the region `Root_Wiper` if the permanent feature is selected for a feature configuration.

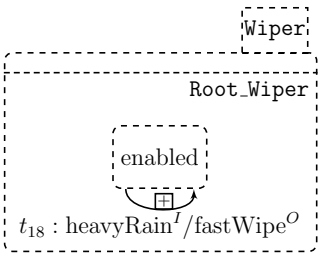
We evolved the original version based on four evolution scenarios resulting in total in five SPL versions $\Theta^{\text{W}} = \{\theta_0^{\text{W}}, \theta_1^{\text{W}}, \theta_2^{\text{W}}, \theta_3^{\text{W}}, \theta_4^{\text{W}}\}$ [NLS18]. The initial higher-order delta $\delta^{H}_{\theta_0^{\text{W}}}$ adds the five deltas shown in Fig. 3.18 such that combined with the core state machine depicted in Fig. 3.17, we obtain the delta model for the initial SPL version θ_0^{W}. The remaining scenarios for the SPL versions θ_1^{W} to θ_4^{W} include (1) the improvement of the rain sensor and wiper by a new detectable rain intensity, (2) the incorporation of rain intensities for permanent wiping, (3) the addition of a window cleaning functionality, and (4) its improvement by a level check for the screenwash liquid. Due to the evolu-



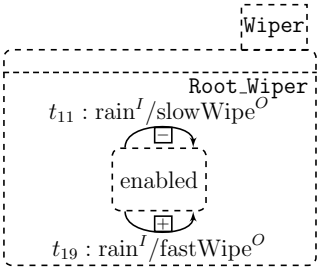
(a) δ_{HS} with r_ϵ = `Root_Sensor` and φ_{HS} = HighQualitySensor && (LowQualityWiper || HighQualityWiper)



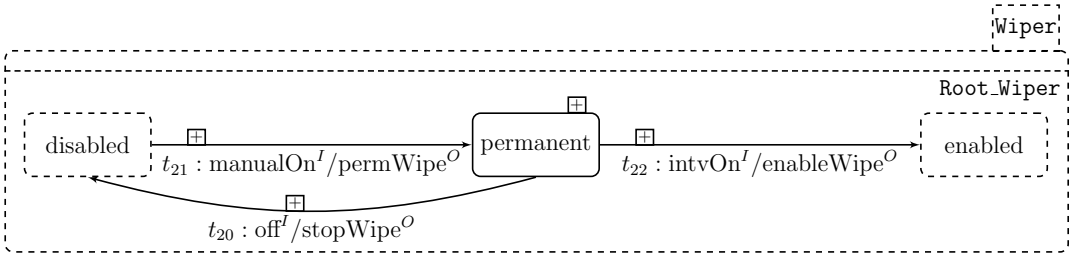
(b) δ_{HSLW} with r_ϵ = `Root_Wiper` and φ_{HSLW} = HighQualitySensor && LowQualityWiper



(c) δ_{HSHW} with r_ϵ = `Root_Wiper` and φ_{HSHW} = HighQualitySensor && HighQualityWiper



(d) δ_{LSHW} with r_ϵ = `Root_Wiper` and φ_{LSHW} = LowQualitySensor && HighQualityWiper



(e) δ_{Perm} with r_ϵ = `Root_Wiper` and φ_{Perm} = Permanent

Figure 3.18: Delta Set of the Initial Wiper SPL Version

Table 3.4: Overview of Key Parameters for the Subject Product Lines **W**iper, **V**ending **M**achine, and **M**ine **P**ump (# = Number, ⌀ = Average)

SPL	#Features	#Variants	Size Core (Regions+States+Transitions)	# Deltas	⌀ Size Variants (Regions+States+Transitions)
W_{θ_0}	8	8	25 (3+8+14)	5	29.5 (3.0+9.0+17.5)
W_{θ_1}	8	8	27 (3+8+16)	9	35.0 (3.0+9.5+22.5)
W_{θ_2}	9	12	27 (3+8+16)	11	40.6 (3.3+11.0+26.3)
W_{θ_3}	10	24	27 (3+8+16)	14	45.6 (3.3+12.0+30.3)
W_{θ_4}	10	24	27 (3+8+16)	15	50.1 (3.8+14.0+32.3)
VM_{θ_0}	9	28	26 (2+12+12)	6	29.2 (2.0+12.5+14.7)
VM_{θ_1}	13	42	26 (2+12+12)	12	33.4 (2.0+13.5+17.9)
VM_{θ_2}	14	70	26 (2+12+12)	18	34.7 (2.0+13.9+18.7)
VM_{θ_3}	14	42	26 (2+12+12)	14	31.7 (2.0+13.2+16.4)
VM_{θ_4}	13	42	26 (2+12+12)	17	42.1 (3.1+17.8+21.2)
VM_{θ_5}	13	42	26 (2+12+12)	18	46.1 (3.7+19.5+22.9)
VM_{θ_6}	13	48	56 (5+25+26)	25	57.7 (4.6+24.2+28.8)
MP_{θ_0}	7	16	45 (5+18+22)	8	65.5 (6.5+25.5+33.5)
MP_{θ_1}	7	16	45 (5+18+22)	9	70.5 (7.0+27.5+36.0)
MP_{θ_2}	8	32	45 (5+18+22)	20	77.2 (7.7+30.0+39.5)

tion, the number of variants increases from eight ($\mathbb{V}_{\theta_0^W}$) over 12 ($\mathbb{V}_{\theta_2^W}$) to 24 for the last version $\mathbb{V}_{\theta_4^W}$. Furthermore, to integrate the functionality of the evolution scenarios, the number of deltas and the size of variant-specific state machines also increase. We summarize the respective key parameters such as number of features, number of variants, size of the core state machine, number of deltas, and average size of variant-specific state machines for each SPL version of the Wiper SPL in Tab. 3.4. In case the size of the core state machine varies between subsequent SPL versions, the respective evolution step has an impact on the core by means of at least one newly added or modified delta which is always applied on the core before transforming it into a variant-specific state machine.

Vending Machine (VM). The SPL defines a family of a control software for vending machines selling hot beverages. It was first proposed by Fantechi and Gnesi [FG08] and was extended by Classen [Cla10]. The SPL allows for the derivation of 28 variants offering at least one of three different beverages, namely coffee, tea, and cappuccino, which are paid either in Euro (€) or US Dollar ($). In addition, an optional ring tone is emitted as soon as a beverage has been delivered. The behavioral specification of the original SPL [Cla10] was modeled via a featured transition system [CCS+13]. Again, as a first step, we remodeled the behavior using delta state machines to become delta-oriented [LMT+16; NLS18]. Afterwards, we altered the SPL based on six evolution scenarios such that its evolution history defines in total seven SPL versions $\Theta^{VM} = \{\theta_0^{VM}, \theta_1^{VM}, \theta_2^{VM}, \theta_3^{VM}, \theta_4^{VM}, \theta_5^{VM}, \theta_6^{VM}\}$ [NLS18]. The consecutive scenarios include (1) the introduction of different sizes of the offered beverages for the European machines in terms of the sizes `small`, `regular`, and `large`, (2) the adaptation of the beverage sizes to become variable selectable for a variant, (3) the removal of the small beverage size, (4) the introduction of a milk counter, (5) its improvement by a display showing the current filling level, and (6) the removal of the restriction that milk is solely selectable for cappuccino. Due to the

evolution, the number of variants increases from 28 ($\mathbb{V}_{\theta_0^{\text{VM}}}$) over 42 ($\mathbb{V}_{\theta_4^{\text{VM}}}$) to 48 for the last version $\mathbb{V}_{\theta_6^{\text{VM}}}$. Again, the update of the variant set results in an increase of the number of deltas and the size of variant-specific state machines. The key parameters for each SPL version of the Vending Machine SPL are summarized in Tab. 3.4, where a varying core state machine size indicates that the core is adapted during the evolution history.

Mine Pump (MP). The SPL realizes the variable control software of a pump system automatically pumping water out of a mine shaft. Kramer et al. [KMS+83] first introduced the Mine Pump system and Classen [Cla10] adapted it as SPL. The SPL defines 16 variants incorporating the variable handling of different levels of incoming water and an optional methane detection facility. In addition, the automated pumping procedure can optionally be controlled by starting and stopping the pump manually via a command console. Again, the behavioral specification of the original SPL [Cla10] was modeled via a featured transition system [CCS+13] which we remodeled using delta state machines to become delta-oriented [LMT+16; NLS18]. We evolved the SPL based on two evolution scenarios such that its evolution history defines in total three SPL versions $\Theta^{\text{MP}} = \{\theta_0^{\text{MP}}, \theta_1^{\text{MP}}, \theta_2^{\text{MP}}\}$ [NLS18]. The evolution scenarios include (1) the introduction of a methane extraction functionality such that methane is also pumped out of a mine shaft, and (2) the introduction of an air level checking facility. Due to the evolution, the number of variants increases from 16 ($\mathbb{V}_{\theta_0^{\text{MP}}}$) to 32 for the last version $\mathbb{V}_{\theta_2^{\text{MP}}}$ also impacting the number of deltas and the size of variant-specific state machines. The key parameters for each SPL version of the Mine Pump SPL are summarized in Tab. 3.4.

3.5 Related Work

In this section, we discuss related work regarding the evolution of domain artifacts in the solution space as well as in the problem space as higher-order delta modeling belongs to this category of SPL evolution techniques. We use the four requirements defined at the beginning of this chapter, namely (1) integrated modeling formalism (**R1**), (2) adaptability (**R2**), (3) evolution history documentation (**R3**), and (4) analysis of evolution (**R4**), to provide a categorization of the discussed techniques and to facilitate a better comparison to each other. The categorization is summarized in Tab. 3.5 at the end of this section. For a general overview on recent techniques for coping with SPL evolution, we refer to the literature [LC13; BP14; MD16]. In addition, we refer to Schaefer et al. [SRC+12] for a survey on variability modeling.

3.5.1 Problem Space Evolution

The evolution of problem space artifacts is mainly tackled on the feature model level [BPD+10; NSS16; TM14; TBK09; BKL+16; DvDP17; QPB+14; GW10; GF11] or on the level of the requirement specification of an SPL [TB07; PYZ11; dOdA15].

Feature Model Evolution. **Botterweck et al.** [BPP+09; BPD+10] proposed the EvoFM approach, where a feature model is defined to capture feature model evolution. An EvoFM feature encapsulates one or more feature model elements and its type denotes the potential influence of evolution, e.g., mandatory EvoFM features are not affected by evolution steps and are constant during the evolution history. Hence, an EvoFM configuration, i.e., a selection and composition of EvoFM features, represents a feature model for a concrete SPL version. **Pleuss et al.** [PBD+12] integrated the EvoFM approach in the EvoPL framework facilitating the strategic planning of long-term SPL evolution, where further rationals for decisions made during the evolution of an SPL are incorporated and

mapped to EvoFM features. They also implemented analyses to ensure the structural consistency of the EvoFM feature model and EvoFM configurations. **Schubanz et al.** [SPB+12; SPP+13] extended EvoPL such that, in addition to the feature model evolution, the decision making process is captured, i.e., each evolution step has its own design decision, change rationals etc. EvoFM/EvoPL is an integrated modeling technique solely applicable in the problem space, where feature modeling is used to capture and document the variability and evolution of an SPL (**R1**,**R3**). Although not discussed by the authors, an EvoFM feature may encapsulate also elements of other variability modeling techniques, e.g., OVM [PBvdL05], and, therefore, has the potential to be adapted for other domain artifacts of the problem space. The provided analyses solely ensure the consistency, but do not allow for, e.g., change impact analysis, as EvoPL focuses on the planning of SPL evolution.

Nieke et al. [NSS16] proposed temporal feature models as extension of standard feature models. To capture SPL evolution, almost every element of a feature model, e.g., features, feature groups, or constraints, is defined as temporal entity and, therefore, can be versioned and further the used constraint language incorporates versions as first-class entity. Based on a given version, a temporal feature model is projected to its version-specific instance similar to annotative modeling techniques [SRC+12; CA05]. In addition to the modeling capability, temporal feature models facilitate the reasoning about the change impact on feature configurations based on a catalog of (atomic/complex) evolution operations and their defined application semantics. The authors apply the impact analysis to support SPL developers to guarantee that the evolution does not affect certain feature configurations of interest, i.e., the configurations remain valid during the evolution step. **Nieke et al.** [NES17] extended their work on temporal feature models by integrating the approach into the tool suite DarwinSPL allowing for the evolution of context-aware dynamic SPLs. Furthermore, **Nieke et al.** [NST18; NMS+18] defined an evolution anomaly detection and explanation technique based on temporal feature models. The technique identifies inconsistencies, the operations that lead to them, and the respective evolution step by investigating the complete evolution history, e.g., to incorporate intermediate evolution steps that do not violate the validity of already planned evolution steps. **Mauro et al.** [MNS+18] integrated temporal feature models and, therefore, DarwinSPL in a framework for context-aware reconfiguration of evolving dynamic SPLs. In the end, temporal feature models can be interpreted as improvement of hyper feature models [SSA13a; Sei17] (cf. Sect. 3.5.3) as, besides the versioning of features, the complete evolution of a feature model is documented in an integrated fashion (**R1**,**R3**). The modeling technique is hard to adapt for other artifact types specifying the problem space. However, **Nieke et al.** [NSS16; NST18; NMS+18] provide various analyses of evolution, e.g., for change impact or anomaly detection, supporting the management of SPL evolution within the problem space (**R4**).

Tran and Massacci [TM14] managed feature model evolution based on two types of models. The first model called evolution possibility model specifies for the features of a feature model their possibilities to evolve during or to be affected by an evolution step. The second model, i.e., the evolutionary feature model, defines the feature model and all changes made by evolution steps. In addition, two types of analyses are realized incorporating both models. Similar to **Nieke et al.** [NSS16], the survivability analysis determines whether the validity of existing feature configurations still holds. In contrast, the repair cost analysis predicts the effort for repairing existing configurations affected by the evolution step to become valid again. As two models are used to capture and document the feature model evolution, the proposed formalism is not completely integrated and

is solely applicable to feature models (**R3**). The defined analyses of evolution provide information to support the SPL developer during evolution (**R4**).

In contrast to the previously discussed techniques, where the evolution is captured in a respective (feature) model, model differencing techniques are applied to determine the change operations made by an evolution step [TBK09; BKL+16; DvDP17]. Those operations are then used to reason about their impact on the variant set. Hence, the evolution is neither modeled nor documented, but an SPL developer is supported by those techniques as he/she gets directly feedback about the impact of the changes which he/she made. **Thüm et al.** [TBK09] proposed a classification of feature model evolution w.r.t. the impact on the set of feature configurations in terms of refactorings, specializations, generalizations, or arbitrary edits. By comparing the original and the evolved feature model, their algorithm detects the applied change operations and computes the respective change classification by applying a constraint solver. **Bürdek et al.** [BKL+16] investigated the evolution of a real-world case study from the automation engineering domain and identifies a catalog of edit operations. They detect changes between two versions of a feature model and document them as a sequence of (complex) edit steps. This sequence is then exploited to compute the semantical difference between both feature model versions w.r.t. the set of derivable variants, where the classification of **Thüm et al.** [TBK09] comes into effect. **Dintzner et al.** [DvDP17] proposed the tool FmDiff and applied it to the Kconfig variability model of the linux kernel to identify change operations which are commonly applied during its evolution history. The detected change operations are then classified via a classification scheme the authors defined. They extended FmDiff by the tool Fever to allow for the identification of change operations also from the build system and source code of the Linux kernel [Din17] and also applied their classification scheme. As a result of their performed studies, they further found out that already existing features and their related source code are most frequently modified for evolving the Kconfig variability model of the Linux kernel.

Quinton et al. [QPB+14] did not apply model differencing techniques, but discussed common change operations applied to evolve cardinality-based feature models based on the investigation of systems from the cloud domain. They further examined the change operations and their relation to resulting inconsistencies. Both, the set of change operations and the information about their impact, are exploited to realize a tool support using off-the-shelf solver for automated detection and explanation of inconsistencies during the evolution of cardinality-based feature models. **Guo and Wang** [GW10; GWT+12] also specified a set of change as well as recovery operations for feature model evolution, but from an ontological perspective. Change operations represent the standard evolution, whereas recovery operations maintain consistency after the application and detection of error-prone operations, e.g., the removal of a feature requires the removal of a related feature to re-ensure the consistency of the evolved feature model. **Gamez and Fuentes** [GF11; GF13] define the evolution of cardinality-based feature models and clonable features and exploit their categorization of changes to automatically propagate those changes to existing feature configurations. Based on the change propagation, an existing configuration is adapted to correspond to the new SPL version, which, in turn, is also used to adapt the product line architecture based on the mapping between architecture elements and features.

Evolution of Requirement Specification. In addition to the problem space evolution on the feature model level, the evolution of the requirement specification of an SPL is tackled in the literature [TB07; PYZ11; dOdA15]. **Thurimella and Bruegge** [TB07] introduced an approach for rationale-

based SPL evolution. They exploited the questions, options and criteria model as well as a modified version of the EasyWinWin strategy to capture and reason about the rationales of change requests of the SPL requirements. **Peng et al.** [PYZ11] proposed a problem-oriented technique for handling SPL evolution. They focused on the (co-)evolution of the SPL domain context and requirement specification to analyze the impact on the variability defined by the features of the feature model based on traceability links between, e.g., requirements and features. The analysis facilitates the future planning and risk management of SPL evolution. **Oliveira and Almeida** [dOdA15] realized a method for feature-driven requirements engineering evolution. In this method, the evolution of the requirements specified by use cases is managed by adapting and applying the safe evolution templates of **Neves et al.** [NBA+15] (cf. Sect. 3.5.3). Furthermore, the traceability matrix linking requirements and features is also updated. Afterwards, the update is exploited in such a way that the changes of requirements are propagated to also evolve the feature model.

Compared to higher-order delta modeling capturing SPL evolution in the solution space, those techniques are solely applicable for managing problem space evolution mainly on the feature model level. Furthermore, only a subset of the discussed techniques provide a modeling formalism to capture and document the variability as well as evolution of an SPL facilitating also the analysis of evolution, e.g., for change impact analysis. We summarize those techniques in Tab. 3.5 by classifying them w.r.t. the four requirements we derived for integrated modeling of SPL evolution. However, as higher-order delta modeling solely captures SPL evolution in the solution space, a combination with one of the discussed techniques, e.g., temporal feature models [NSS16], is reasonable to tackle evolution in both spaces.

3.5.2 Solution Space Evolution

The evolution of solution space artifacts is mainly managed by adopting the concepts of variability implementation techniques [AMC+05; AMC+07; HRR+12; SSA14a; KLL+14; ALR+05; LKS16; LNT+18].

In addition to higher-order delta modeling (cf. Sect. 3.3), we also proposed an extension for the annotative implementation technique 150% modeling [SRC+12; CA05] called *175% modeling* [LNT+18; LRB+19]. 175% modeling [LNT+18; LRB+19] merges all models of variants and versions of variants in one super model. In this superimposed model, elements are annotated with combined feature and version annotations to specify their potential containment in the respective model of a variant or its versions. Similar to delta modeling [CHS15; Sch10] as a transformational variability implementation technique [SRC+12], annotative techniques [SRC+12; CA05] are also well-suited to capture and, therefore, document the variability and the evolution of SPLs by the same means (**R1**,**R3**). Just as higher-order delta modeling, 175% modeling denotes an integrated modeling formalism adaptable for various types of domain artifacts [CA05; CAK+05; ABK+16; CCS+13; MPC16; GLS08] (**R2**) and applicable to support SPL quality assurance [CCS+13; MPC16; GLS08; LBL+14; Loc13; FSM18; BLB+15; COL+11; DPL+14; DPL+15; tBdVW17; LRB+19; LKL12; OWE+11; ER11] by facilitating family-based analysis [TAK+14] also for evolving SPLs. Although both modeling formalisms are well-suited to be applied for test modeling of evolving SPLs, i.e., both formalisms fulfill the requirements **R1**, **R2**, and **R3**, the distinct types of analysis supported by the formalisms are the main differentiating factor. 175% modeling focuses on the commonality of variants and versions of variants by merging the respective models in a superimposed model. Based on the superimposition, the application of family-based analyses [TAK+14] is facilitated such that a reasoning about test-case reusability as well

as efficient generation of test cases for variants and versions of variants is achievable. In contrast, higher-order delta modeling focuses on the differences between variants and versions of variants by capturing the differences as (higher-order) deltas. Based on this explicit specification of differences, the application of change impact analysis is facilitated (cf. Chapt. 4). As change impact analysis is crucial for regression testing [YH12], e.g., to guide retest test selection, we select higher-order delta modeling to be applied as test-modeling formalism in this thesis. Hence, we omit the definition of 175% modeling as we integrate higher-order delta modeling in our model-based regression testing framework for evolving SPLs (cf. Chapt. 5). However, we showed the equivalence of both formalisms in terms of the derivable variant- as well as version-specific models and provided a bi-directional transformation between them representing a variant-preserving mapping according to Fenske et al. [FTS13] in order to exploit their benefits if solely one formalism is applicable [LNT+18].

Haber et al. [HRR+12] adapted delta modeling [CHS15; Sch10] for capturing the SPL evolution of software architectures defined in the architecture description language MontiArc. They alter a delta model by adding, removing, or modifying deltas, but do not record the evolution operations required to document the evolution history. In addition, they classify the result of SPL evolution based on three scenarios such that either variants are added, removed, or modified during an evolution step. To ensure the consistency of a delta model version, **Haber et al.** [HRR+12] exploited a family-based analysis implemented in the MontiArc framework and further presented refactorings just as Schulze et al. [SRS13]. Based on the refactorings, they prevent from the degeneration of the delta model which is a problem according to Gaia et al. [GFF+14]. Compared to higher-order delta modeling, their approach is very similar as they provide the same evolution operations on the delta set of an delta model. However, they do not capture those operations, whereas we lift the notion of delta modeling [CHS15; Sch10] to specify the transformation and, therefore, to explicitly capture the evolution of delta models of respective SPL versions. Although specified for MontiArc, their technique is potentially applicable for other types of solution space artifacts based on the adaptation of delta modeling (**R2**). The provided analysis solely allows for ensuring the consistency of a delta model, but there is no analysis of the evolution, e.g., for change impact analysis, defined.

Kowal et al. [KLL+14] applied delta modeling [CHS15; Sch10] to define a multi-perspective modeling method for evolving SPLs in the automation domain. They adapted delta modeling for each of their perspectives, namely the system workflow, system architecture, and component behavior in terms of state machines. Similar to **Seidl et al.** [SSA14b; Sei17] (cf. Sect. 3.5.3), they specify variant- as well as version-specific deltas to capture the variability and evolution of an SPL on the same modeling level such that a designated core is either transformed in a variant or a version of a variant. The proposed method is integrated as the variability and evolution of an SPL are captured by the same means (**R1**), but the evolution history is not documented. Again, delta modeling [CHS15; Sch10] facilitates the application for other artifact types (**R2**). In contrast to higher-order delta modeling, no analysis of the evolution is provided.

Lima et al. [LSK+13] proposed a delta-oriented method for managing SPL evolution. They focus on the evolution scenario that an SPL independently evolves in different branches which have to be merged back to the master branch. Therefore, their method identifies conflicts between development branches of an SPL by applying differencing techniques and record the differences as a delta model. In this context, a delta model encapsulates the information what is added, modified, or removed between the branches under consideration w.r.t. the identified conflicts. The obtained delta

model is then exploited to facilitate a semi-automatic resolution of the conflicts and merging of the development branches. Compared to higher-order delta modeling, this method applies deltas solely for capturing differences between versions. The variability of an SPL is realized based on other variability implementation techniques [SRC+12]. Hence, the proposed method is not integrated as evolution and variability are handled differently, but their technique is adaptable for different artifact types for which differencing techniques are applicable. Furthermore, the evolution history is not documented and no analysis of the evolution is defined.

Alves et al. [AMC+05; AMC+07] used aspect-oriented programming (AspectJ) to handle SPL evolution also supporting extractive as well as reactive SPL development. To implement the changes of an evolution step, either new aspects are created and added to the set of aspects, removed from the set, or already defined aspects are modified. In this context, **Alves et al.** [AMC+05; AMC+07] do not distinguish between newly added aspects and already existing aspects, and hence, do not incorporate version information in the evolution process. Every SPL version is defined by its core, which can also be modified, and a set of aspects. Compared to higher-order delta modeling, the technique provides no integrated formalism as variability and version information are not incorporated as first-class entities. The set of aspects is altered during an evolution step, but those changes are neither documented nor analyzed to reason, e.g., about their change impact. The technique is applied to source code, but the aspect-oriented paradigm is also instantiated for UML design modeling [EAB02] and, thus, is adaptable to other types of domain artifacts (**R2**).

Apel et al. [ALR+05] proposed the combination of feature-oriented and aspect-oriented programming to manage SPL evolution. They emphasized the problems of feature-oriented programming with software evolution, e.g., crosscutting modularity, and, therefore, integrated concepts of aspect-oriented programming, i.e., multi mixins, aspectual mixins and aspectual mixin layers, to solve those problems using FeatureC++ as example. Similar to **Alves et al.** [AMC+05; AMC+07], they do not incorporate version information in their technique, i.e., every SPL version is defined by a set of feature modules implemented as mixin layers in the AHEAD tool suite and a set of aspects if necessary. Hence, new aspects are added or modified to implement the changes of an unanticipated evolution step. Due to the combination of two variability realization techniques as well as the missing incorporation of version information, the presented approach does not allow for an integrated variability and evolution handling. The evolution is not documented and no analysis is applied to support an SPL developer. Similar to higher-order delta modeling, feature-oriented as well as aspect-oriented programming are adaptable for other types of solution space artifacts besides source code facilitating the application of the technique also in another development context (**R2**).

Schach and Tomer [ST00] applied two types of models to capture the evolution of SPLs. First, the evolution-tree model specifies the traceability between requirements, design artifacts, and development artifacts and their respective versions such that a change to the requirements infers the set of artifacts which have to evolve correspondingly. Second, the propagation graph model relates interdependent artifacts and, therefore, the change of an artifact points to further artifacts potentially influenced. Compared to higher-order delta modeling, their technique is not integrated as two models are required to capture the variability as well as evolution of an SPL and no analysis is provided, but the evolution history is well-documented (**R3**). Both models are independent from the modeling/implementation language the artifacts are defined with and, hence, the technique is adaptable to different types of domain artifacts (**R2**).

In addition to the discussed techniques which are explicitly proposed to manage SPL evolution, the applicability of standard variability implementation techniques [SRC+12] for solution space evolution is assessed in literature [FCS+08; DVG+17; GFF+14; FGF+14; AKE12; HWE17]. **Diniz et al.** [DVG+17] examined the applicability of delta-oriented programming to manage SPL evolution. **Hamza et al.** [HWE17] als incorporated delta-oriented programming in its investigation of unanticipated evolution of SPLs. **Figueiredo et al.** [FCS+08] performed a comparative study regarding the design stability by examining the application of aspect-oriented programming and conditioned compilation, i.e., an annotative variability implementation technique, for evolving SPLs. **Abdelmoez et al.** [AKE12] assessed the maintainability of evolving SPLs realized by object-oriented and aspect-oriented programming. **Gaia et al.** [GFF+14] investigated and compared the application of four variability implementation techniques for managing SPL evolution, namely aspectual feature modules, i.e., a combination of aspect-oriented and feature-oriented programming, conditional compilation, feature-oriented programming, and aspect-oriented programming. **Ferreira et al.** [FGF+14] compared feature-oriented programming and conditional compilation. In the end, all assessments take the source code of an SPL as domain artifact into account and determine the impact on the design modularity as well as stability of the SPL code base in the context of change requests to be managed. The obtained results indicated that existing implementation techniques [SRC+12] provide a good starting point for handling SPL evolution with distinct benefits and limitations, but new techniques, e.g., by extending the existing techniques, are required for tackling SPL evolution efficiently and effectively.

3.5.3 Combined Problem and Solution Space Evolution

In contrast to solely tackle either solution or problem space evolution, there are some techniques for complete evolution management of both spaces [Sei17; DNG+08a; NBA+15; Hol12].

Seidl et al. [SSA13a; SSA14b; Sei17] proposed an integrated management of SPL evolution in the problem as well as solution space [SSA14b; Sei17] based on the combination of hyper feature models [SSA13a; SSA14b; Sei17] and an extension of delta modeling [CHS15; Sch10]. Hyper feature models capture the evolution on the feature model level, where features are attributed by versions and the definition of constraints is extended to be version-aware. The delta modeling extension defines two types of deltas which are applied on the same modeling level. On the one hand, configuration deltas capture the variability of an SPL, i.e., the delta adds, removes, or modifies the original functionality of a respective feature to which the configuration delta is also mapped. On the other hand, evolution deltas specify the evolution of an SPL, i.e., the delta modifies the original functionality to correspond to the version-specific functionality of a feature version to which the evolution delta is associated. Both, features as well as feature versions are specified in the corresponding hyper feature model. In this context, **Seidl et al.** [SHA12] also introduced the tool FeatureMapper that maintains the consistency of feature models and feature mappings to solution space artifacts, e.g., deltas, during evolution. Hyper feature modeling denotes an integrated modeling formalism to capture the evolution and the variability of an SPL (**R1**). The evolution history is documented solely for individual features based on version attributes. However, the evolution of the feature model is intentionally not captured [Sei17], but is required for a comprehensive management of SPL evolution. In contrast, similar to higher-order delta modeling, they define an extension of delta modeling [CHS15; Sch10] to capture the variability as well as the evolution of an SPL by the same means (**R1**). Their approach

specifies the evolution on the same level as the variability such that evolution deltas directly transform a model to correspond to the feature version, whereas our approach specifies the evolution and the variability of an SPL on different levels. Based on delta modeling [CHS15; Sch10], their integrated modeling formalism is also adaptable for other solution space artifacts (**R2**). In constrast to higher-order delta modeling, where the evolution history is documented based on a respective delta model, **Seidl et al.** requires the combination with hyper feature models [SSA13a; SSA14b; Sei17] for history documentation. Furthermore, their approach does not provide an analysis of the evolution.

Dhungana et al. [DNG+08a; DNG+08b; DGR+10] introduced the Dopler framework for managing SPL evolution. In Dopler, model fragments are the main entities capturing the variable domain assets of a specific part of an SPL, e.g., components, which are created and altered by an SPL developer. A variability model of an SPL version, from which a specific variant version is derivable, is obtained by selecting and merging a set of model fragments. The merge process is semi-automated as conflicts may occur such as name mismatches to be solved by the developer. To detect inconsistencies, **Dhungana et al.** [DNG+08a; DNG+08b; DGR+10] implemented a change impact analysis that check the consistency after a model fragment is created or maintained. Furthermore, the framework is also applicable for problem space evolution as the variable domain assets captured by model fragments can represent features and their interrelation. **Heider et al.** [HRG12] extended Dopler with the Puple framework that supports the propagation of the evolution changes to already derived and operating products in terms of updates. The Dopler framework incorporates variability and version information as first-class entities, i.e., model fragments capture and document variable domain assets to be merged for a specific SPL version to define the respective variability model. However, model fragments may also evolve and those evolution changes are not explicitly specified and, therefore, not documented as in higher-order delta modeling. The ability of model fragments to capture different types of domain assets shows the adaptability of the framework (**R2**). In addition, a change impact analysis is realized to provide an SPL developer feedback about potentially introduced inconsistencies, but the impact of the changes on the variant set is not examined.

Neves et al. [NTS+11; NBA+15] proposed templates for the safe evolution of SPLs, where they focus on refactorings and extensions of SPLs as evolution scenarios. The templates facilitate the evolution of feature models, domain artifacts, and their mapping and ensures that the functionality of the original SPL is preserved. **Borba et al.** [BTG12] extended the work by providing a formal foundation in terms of a SPL refinement theory. The theory facilitates the verification that the application of the evolution templates assure behavior preservation. **Ferreira et al.** [FBS+12] developed a tool set for the application as well as analysis of the safe evolution templates. The analysis checks whether an applied template results in a refinement in order to reason about behavior preservation. **Teixeira et al.** [TAB+15] further extended the work of **Borba et al.** [BTG12] by introducing a product line of refinement theories. They investigated the commonalities and differences between various instantiations of the refinements implemented by evolution templates for modeling/programming languages and exploited the results for the product line definition. In this context, a feature denotes the host language or specific templates and a respective feature configuration specifies a theory instantiation. The product line facilitates the reuse of template specifications and the extension by new languages. **Sampaio et al.** [SBT16] proposed partially safe evolution to also incorporate modifications and removals as evolution scenarios. They verify behavior preservation for those variants which do not contain a modified or removed artifact. Therefore, the analysis results are usable to

reason about change impact to provide an SPL developer the information which variants are not affected by changes. Compared to higher-delta modeling, safe evolution templates represent no integrated modeling formalism as they define how to apply changes of an evolution step in order to represent a refinement and not the operations itself. In addition, the evolution history is not explicitly captured. Based on the refinement theory of **Borba et al.** [BTG12] and the extension by **Sampaio et al.** [SBT16], safe evolution templates and their application facilitate a change impact analysis to support SPL developers (**R4**). The analysis provides as result the set of new and modified variants, whereas higher-order delta modeling provides in addition how a modified variant has changed compared to its previous version. The work of **Teixeira et al.** [TAB+15] shows the adaptability of the concept of safe evolution templates (**R2**).

Holdschick [Hol12] proposed how variability and evolution are incorporated in the automotive domain using functional blocks and feature modeling as modeling formalisms for capturing the variability of an SPL. In addition, he described three potential evolution scenarios, e.g., the introduction of a new alternative component, and what steps are required to evolve the feature as well as functional block model to correspond to the next version. Compared to higher-order delta modeling, the approach does not provide a modeling technique to cope with SPL evolution, but solely how to react on changing requirements. Furthermore, the evolution history is not captured, no analysis is given, and the approach is specific for functional block modeling.

Table 3.5: Categorization of Related Work on SPL Artifact Evolution (Fulfilled = +, Partially Fulfilled = ○, Unfulfilled = −, S = Solution Space, and P = Problem Space)

Approach	Variability Space	Integrated Modeling (**R1**)	Adaptability (**R2**)	Evolution History (**R3**)	Analysis of Evolution (**R4**)
Higher-Order Delta Modeling	S	+	+	+	○
EvoPL [BPD+10; PBD+12; SPP+13]	P	+	○	+	○
DarwinSPL [NSS16; NES17; NMS+18]	P	+	−	+	+
Tran and Massacci [TM14]	P	○	−	+	+
175% Modeling [LNT+18; LRB+19]	S	+	+	+	○
Haber et al. [HRR+12]	S	−	+	−	○
Kowal et al. [KLL+14]	S	+	+	−	−
Lima et al. [LSK+13]	S	−	○	−	−
Alves et al. [AMC+05; AMC+07]	S	−	+	−	−
Apel et al. [ALR+05]	S	−	+	−	−
Schach and Tomer [ST00]	S	−	+	+	−
Seidl et al. [SSA13a; SSA14b; Sei17]	P/S	+	+	○	−
Dopler [DNG+08a; DNG+08b; DGR+10]	P/S	○	+	○	○
Evolution Templates [NBA+15; TAB+15; SBT16]	P/S	−	+	−	+
Holdschick [Hol12]	P/S	−	−	−	−

3.6 Chapter Summary

Test modeling is a foundational step for applying model-based testing techniques [UL06; UPL12; LPK+14]. For the definition of our model-based regression testing framework (cf. Chapt. 5) for testing evolving SPLs, we, therefore, require a respective formalism that allows for the specification of

the behavior of variants and version of variants. To this end, we proposed higher-order delta modeling as new variability implementation technique applicable not solely in the context of quality assurance, but also for model-driven engineering of evolving SPLs. Higher-order delta modeling extends delta modeling [CHS15; Sch10] such that higher-order deltas transform version-specific delta models, e.g., representing the variable test model for one SPL version in time, to correspond to the delta model of the subsequent SPL version by altering the encapsulated delta set via additions, removals, and modifications of deltas. Furthermore, the evolution history is documented by means of higher-order delta models. In this thesis, we instantiated (higher-order) delta modeling for state machines as state machines are a well-established modeling technique for model-based testing [UL06; Wei10; LPK+14; Loc13; LTW+14]. However, due to the artifact-independency of delta modeling [CHS15; Sch10], our extension is also adaptable for various types of solution space domain artifacts, e.g., software architectures [LLL+14; LLL+15; HRR+12]. As a further benefit, our modeling formalism facilitates change impact analysis by analyzing the application of higher-order deltas (cf. Chapt. 4), where the evolution impact in terms of new, unchanged, or modified variants is provided to be exploited by guiding our regression testing framework (cf. Chapt. 5). We realized a prototypical tool support called Dope for capturing the evolution of SPLs via higher-order delta modeling and applied it to three model-based SPLs which are used as subject systems for the evaluation of our change impact analyses in Chap. 4 and of our regression testing framework in Chap. 5.

4 Delta-Oriented Change Impact Analysis

The content of this chapter shares material with work published in [LBS15], [LKS16], [LMT+16], and [LNT+19].

Contribution

We propose two change impact analysis techniques allowing for an automated reasoning about retest potentials between consecutively tested variants as well as between product-line versions under test. We exploit the explicit specification of differences between variants and versions of variants captured by (higher-order) delta modeling. First, we introduce incremental model slicing facilitating the automated identification of changed execution dependencies, i.e., behavior potentially influenced by changes to the test model between subsequent variants and also between modified versions of variants. Second, we reason about the application of higher-order deltas and deduce the resulting changes to the variant set when stepping to the next product-line version under test in terms of added, removed, modified, and unchanged versions of variants. We prototypically implement our change impact analyses and evaluate their applicability and efficiency based on the three evolving model-based subject software product lines.

In this chapter, we introduce change impact analysis techniques applied in our model-based regression testing framework (cf. Chapt. 5). Change impact analysis is crucial for regression testing to guide retest test selection and prioritization strategies [YH12]. By examining the differences, i.e., applied changes, between variants or versions of variants, we are able to identify the impact of the differences to already tested behavior by means of changed (inter)dependencies of software parts indicating retest potentials to be revalidated during regression testing [YH12]. Furthermore, when stepping to the next SPL version under test, we are interested in the information, whether a variant gets modified or stays unchanged. Based on this information, we are able to guide the retest of versions of variants and increase the reduction of the overall testing effort for evolving SPLs.

Program slicing [Wei81; Tip95] is a promising technique already applied for static and dynamic analysis in the context of single-software testing [AHK+93; JG06; BH93; RH94; GHS96; Bin97; Bin98; TLS+10; AR11; HHD99; HHH+02; HD95; HHF+02], e.g., for change impact analysis supporting white-box regression testing [AHK+93; JG06; BH93; RH94; GHS96; Bin97; Bin98; TLS+10; AR11]. Those regression testing techniques proposed different types of static and dynamic slices w.r.t. the execution of test cases. If a slice for a test case contains a modified statement, the test case is selected for its reexecution. In this thesis, we propose model-based regression testing of evolving SPLs (cf. Chapt. 5) and focus on slicing applied for change impact analysis. However, the existing slicing techniques [AHK+93; JG06; BH93; RH94; GHS96; Bin97; Bin98; TLS+10; AR11] are not applicable to support our testing framework. First, those techniques are solely program-based, whereas

our model-based framework uses state machines as test-modeling formalism. Second, in most cases, a slice is computed dynamically w.r.t. the execution of a test case. In contrast, we are interested in the static dependencies and their changes for a given slicing criterion, e.g., a transition, used for the slice computation. The subsequent process of retest test selection is then similar to the existing techniques, i.e., a test case is selected for a retest, if its execution traverses the changed dependency (cf. Chapt. 5). Third, those techniques are proposed for single-software systems and would result in a redundant and, hence, inefficient analysis due to the shared commonality between variants of an SPL. For these reasons, we require a slicing technique that (1) is applicable to state machine (test) models, (2) is capable to detect changed dependencies representing the impact of changes to a state machine indicating retest potentials, and (3) takes the explicit knowledge about commonality and variability into account to allow for efficient SPL analysis.

In this thesis, we propose incremental model slicing and its application for change impact analysis to guide the retest test selection of our regression testing framework. Our technique exploits the commonality between variants and versions of variants and focuses on the differences explicitly specified by state machine regression deltas. We use those differences to facilitate the incremental computation of a slice for a given slicing criterion based on an existing slice for the same criterion, yet computed for a previously analyzed variant. By incorporating the previous slice, we are able to determine the differences between both slices. The slice differences indicate (unintentional) changes of the execution dependencies and, therefore, refer to potential behavior to be retested.

Furthermore, we are interested in how the variant set changes when stepping to the next SPL version under test in terms of added, modified, and unchanged variants. Existing techniques providing such kind of change impact analysis [NSS16; SBT16; TBK09; BKL+16] either detect changes to the variant set on the feature model level [NSS16; TBK09; BKL+16] or identify solely that a variant is modified [SBT16], but do not determine how the original version and the modified version of the variant differ. In contrast, we exploit the application of higher-order delta modeling as test-modeling formalism to reason about changes to the variant set between consecutively tested SPL versions. Higher-order deltas specify how the delta set of a version-specific delta model changes in terms of additions, removals, and modifications of state machine deltas. We take those changes into account to infer and reason about the respective changes on variant-specific delta sets resulting in a categorization in terms of added, modified, and unchanged variants. Based on this categorization, our regression testing framework (cf. Chapt. 5) is guided such that modified variants are tested based on their previous version and unchanged variants are skipped as they are already tested.

The remainder of this chapter is structured as follows. First, we describe incremental model slicing and its application as change impact analysis in Sect. 4.1. Second, we introduce the reasoning about the application of higher-order deltas and its impact on the set of variants in Sect. 4.2. Third, we evaluate the applicability as well as efficiency of our change impact analysis techniques and discuss the results in Sect. 4.3. Fourth, we discuss related work on slicing and change impact analysis in the context of regression testing and SPLE in Sect. 4.4. Finally, we conclude the chapter in Sect. 4.5.

4.1 Change Impact Analysis of Variants

In this section, we describe incremental model slicing and the identification of changed dependencies which is exploited by our model-based regression testing technique to guide the retest test selection (cf. Chapt. 5). Our technique represents an extension of model slicing [ACH+13] applicable

for delta state machine test models that exploits the commonality between variants and versions of variants and focuses on the differences explicitly specified by deltas. In the following, we first describe shortly the foundations of slicing and afterwards introduce our incremental technique as well as how it is applied as change impact analysis.

4.1.1 Program and Model Slicing

In the early 1980s, Weiser [Wei81; Wei84] introduced *program slicing* as a static analysis technique for procedural programs. Its first application scenario was software maintenance to support developers, e.g., during debugging [Wei82; Tip95]. Since then, the range of application scenarios increased [Tip95; Luc01; BH04; XQZ+05; Sil12] from program comprehension [LFM96] over regression testing [Bin98] to software metrics [OT93] and mutation testing [HHD99]. Based on a program statement and a set of variables defining a *slicing criterion*, the original program is reduced by removing and, therefore, abstracting from statements that do not have an influence on the criterion. The reduced program is called *program slice* and preserves the execution semantics of the original program w.r.t. the given slicing criterion. The computation of a program slice is achieved *statically* or *dynamically*. A static slice incorporates all statements which potentially influence the slicing criterion. In contrast, a dynamic slice contains all statements that are traversed and, hence, influence the slicing criterion during execution based on an initial valuation of the input variables of the program to be sliced. By taking an initial input valuation into account, the slice computation may abstract from more statements resulting in a smaller slice compared to its static version [Tip95].

Independent from the applied strategy, a slice is computed either backwards or forwards. For *backward slicing*, we start from the slicing criterion and analyze backwards which statements have an influence on the criterion and, therefore, are added to the slice until the program's entry point is reached, e.g., the function definition. In contrast, *forward slicing* also starts from the slicing criterion, but analyzes forwards which statements have to be added to the slice as they are influenced by the slicing criterion until the program's exit point is reached, e.g., return statement of a function. Both directions have their applications such as debugging for backward slicing and program comprehension for forward slicing [Tip95; Luc01; BH04; XQZ+05; Sil12]. We refer the reader to Binkley et al. [BDG+06] for a theoretical investigation of program slicing. For a general overview about program-slicing techniques, we refer to respective surveys [Tip95; Luc01; BH04; XQZ+05; Sil12].

Moreover, due to the increasing application of model-driven engineering [VSB+06], the concept of program slicing was adapted for state-based models such as state machines [ACH+13]. Similar to progam slicing, *model slicing* [ACH+13] facilitates the reduction of a model, which is in our case a state machine, by abstracting from model elements, i.e., regions, states, and transitions, that do not influence a given slicing criterion *sc*. For state machines, a state or a transition is used as criterion to compute a *state machine slice*. For model slicing, the same computation strategies and directions as for program slicing are applicable [ACH+13]. Therefore, a state machine slice is computed, e.g., statically or dynamically via backward or forward slicing and preserves the execution semantics of the original state machine w.r.t. the given state machine element used as slicing criterion. Typical application scenarios of model slicing are model comprehension and the support of model checking as well as testing [ACH+13].

In this thesis, we focus on backward slicing similar to the program slicing techniques already applied for change impact analysis to support white-box regression testing of single-software sys-

tems [AHK+93; JG06; BH93; RH94; GHS96; Bin97; Bin98; TLS+10; AR11]. Based on backward slicing, we are able to identify the execution dependencies for a slicing criterion, i.e., state machine elements that influence the execution of, e.g., a transition which is used as slicing criterion. A change of those execution dependencies potentially introduced due to unintended side-effects indicate retest potentials in order to validate that the already tested execution of the slicing criterion is not erroneously influenced. In addition, we statically compute a slice as we are interested in all state machine elements that influence a given criterion during execution to reason about changed dependencies when the original state machine has changed. By incorporating the influencing behavior, we are able to determine the complete impact of changes applied to the state machine test model and are not restricted to test-case-specific slices [AHK+93; JG06; BH93; RH94; GHS96; Bin97; Bin98; TLS+10; AR11]. In contrast, by applying forward slicing starting in a changed state machine element, we would solely identify those state machine elements that are reachable and, hence, influenced by the change, but we would not get the information how the changed element is reached from the entry point of the state machine model. As we will apply slicing for every state machine element, e.g., every transition is used as slicing criterion, we are provided with (1) the information which elements are influenced by a change and (2) how the changed element is traversable to check that the slicing criterion is not erroneously influenced. Furthermore, based on backward slicing, we are able to identify all influencing changes for a slicing criterion. Therefore, backward slicing facilitates an easier reasoning about change impact in order to guide the selection of test cases for a reexecution to check for erroneous influences of changed elements.

Based on this decision, a state machine slice represents a well-formed state machine (cf. Def. 3.8) comprising the reduced behavioral specification of a system w.r.t. a given slicing criterion. We require a slice to correspond to a well-formed state machine to allow for the preservation of the state machine semantics.

Definition 4.1: State Machine Slice

A *state machine slice* $slice_{sc}^{sm} = sm' = (R_{sm'}, r_0, \psi_{sm'}, \chi_{sm'}, E_{sm'})$ of the original state machine $sm = (R_{sm}, r_0, \psi_{sm}, \chi_{sm}, E_{sm})$ for a slicing criterion sc is a well-formed state machine, where the following holds:

- $R_{sm'} \subseteq R_{sm}$, i.e., the set of regions is a subset of the original region set,
- $r_0 \in R_{sm'} \bigcap R_{sm}$, i.e., the slice and the original state machine share the root region r_0,
- $\psi_{sm'} : S_{R_{sm'}} \rightarrow \mathcal{P}(R_{sm'})$, i.e., the sub-hierarchy function is updated w.r.t. the original function ψ_{sm} such that $\forall s \in S_{R_{sm'}} \bigcap S_{R_{sm}} : \psi_{sm'}(s) = \psi_{sm}(s) \bigcap R_{sm'}$,
- $\chi_{sm'} : R_{sm'} \setminus \{r_0\} \rightarrow R_{sm'}$, i.e., the parent-hierarchy function is updated w.r.t. the original function χ_{sm} with $\forall r \in R_{sm'} \setminus \{r_0\} \bigcap R_{sm} : \chi_{sm'}(r) = \chi_{sm}(r) \bigcap R_{sm'} \neq \emptyset$, and
- $E_{sm'} = (E_I^{sm'} \cup E_\tau^{sm'} \cup E_O^{sm'}) = \bigcup_{r \in R_{sm'}} E_r \subseteq E_{sm}$, i.e., the event set is a subset of the original event set with $E_I^{sm'} = \bigcup_{r \in R_{sm'}} E_I^r$, $E_\tau^{sm'} = \bigcup_{r \in R_{sm'}} E_\tau^r$, and $E_O^{sm'} = \bigcup_{r \in R_{sm'}} E_O^r$.

Similar as for the delta application (cf. Sect. 3.2), the set of events $E_{sm'}$ of the state machine slice $slice_{sc}^{sm} = sm'$ to be computed are automatically determined at the end of the slicing process.

Control Dependency Analysis. To determine if a state machine element influences another element or is influenced by it, a dependency analysis has to be performed on the original state machine.

The result of this analysis is captured in a *dependency graph*. A dependency graph is a directed graph, where state machine elements, i.e., states and transitions, are contained as *graph nodes* and dependencies between two elements are represented by *directed labeled edges* between their element nodes. The direction of the dependency edge indicates which element is dependent from which element, i.e., the source node is dependent from the target node, and the label defines what kind of dependency exists between the elements. Depending on the dependencies, element nodes may be not connected to other nodes as their state machine elements are not dependent from another element.

Definition 4.2: State Machine Dependency Graph

Let $\mathcal{L}_{Dep}$ be the set of dependency labels representing the names of dependencies applied for the dependency analysis. A *state machine dependency graph* $DG_{sm} = (N_{sm}, Dep_{sm})$ is a directed graph defined over the label set $\mathcal{L}_{Dep}$, where

- $N_{sm} = N_{S_{R_{sm}}} \bigcup N_{T_{R_{sm}}}$ is a finite set of *element nodes* corresponding to state machine elements of *sm*, i.e., $\forall s \in S_{R_{sm}} : n_s \in N_{S_{R_{sm}}}$ as well as $\forall t \in T_{R_{sm}} : n_t \in N_{T_{R_{sm}}}$ holds, and
- $Dep_{sm} \subseteq N_{sm} \times \mathcal{L}_{Dep} \times N_{sm}$ is a labeled *dependency edge relation.*

In literature [KLB12; ACH+13], various control as well as data dependencies are defined affecting the slice computation. In this thesis, we focus on the control dependencies *synchronization dependency* (*sd*), *transition control dependency* (*tcd*), *global control dependency* (*gcd*), and *refinement control dependency* (*rcd*) proposed by Kamischke et al. [KLB12]. The selection of those dependencies is sufficient for the definition of our incremental model slicing technique to be applied for change impact analysis in our model-based SPL regression testing framework. The reasons are as follows:

(1) Based on our event-based state machine dialect to which our slicing technique is applied to, we abstract from data dependencies.
(2) Those dependencies incoporate the hierarchy and concurrency of our state machine dialect.
(3) We apply our state machine dialect as test-modeling formalism, where input events play an important role to specify the abstract input-output behavior of a system under test. The role of input events is also covered by the dependencies.

The control dependencies $\mathcal{L}_{Dep} = \{sd, tcd, gcd, rcd\}$ are defined as follows [KLB12]:

- **Synchronization Dependency** (*sd*) – Two transitions $t = (s', l_t = (e, \{\ldots\}), s'') \in T_r$ and $t' = (s''', l_{t'} = (e', \{e, \ldots\}), s'''') \in T_{r'}$ which are contained in concurrent regions $r, r' \in R_{sm}$ of a state machine *sm* are *synchronization dependent*, i.e., t is dependent on t' (t, sd, t'), if one generates an internal event $e \in E_\tau$ via broadcast during a state machine step $st_{e'} = (C, \{t', \ldots\}, C')$ which the other one consumes as triggering event in the subsequent step $st_e = (C', \{t, \ldots\}, C'')$ of a state machine run $smr = (\ldots, st_{e'}, st_e, \ldots)$.
- **Transition Control Dependency** (*tcd*) – Two transitions $t = (s', l_t = (e, \{\ldots\}), s'')$ and $t' = (s''', l_{t'} = (e', \{e, \ldots\}), s')$ which are sequentially traversed on a state machine path $\rho_{sm} = (\ldots, T_{t'}, T_t, \ldots) \in T_R^*, t' \in T_{t'}, t \in T_t$ either contained (1) $t, t' \in T_r$ in the same same region $r \in R_{sm}$, or (2) $t \in T_r \wedge t' \in T_{r'}$ in hierarchically distinct regions $r, r' \in R_{sm}$ of a state machine *sm* are *transition control dependent*, i.e., t is dependent on t' (t, tcd, t'), if one generates an internal event $e \in E_\tau$ via broadcast during a state machine step $st_{e'} = (C, \{t', \ldots\}, C')$ which the other one consumes as triggering event in the subsequent step $st_e = (C', \{t, \ldots\}, C'')$ of a state machine run $smr = (\ldots, st_{e'}, st_e, \ldots)$.

- **Global Control Dependency** (*gcd*) – A state $s \in S_r$ and a transition $t = (s, l_t = (e, \{\ldots\}), s') \in T_r$ which are contained in the same region $r \in R_{sm}$ of a state machine *sm* are *globally control dependent*, i.e., t is dependent on s (t, gcd, s), if the state is the source of the transition which is triggered by an input event $e \in E_I$ during a state machine step $st_e = (C, \{t, \ldots\}, C')$ of a state machine run $smr = (\ldots, st_e, \ldots)$.
- **Refinement Control Dependency** (*rcd*) – Two states $s' \in S_r$ and $s \in S_{r'}$ which are contained in different hierarchical regions $r, r' \in R_{sm}$ of a state machine *sm* are *refinement control dependent*, i.e., s is dependent on s' (s, rcd, s'), if one is an initial state $s = s_0^{r'}$ and the other one its parent state $r' \in \psi^*(s')$.

The dependencies synchronization dependency and transition control dependency are very similar as they both reason about the synchronization of transitions, but they differ in their focus. The synchronization dependency captures dependencies between *concurrent regions*, whereas the transition control dependency focuses solely on the *hierarchy of regions*. For the descriptions, we incorporated both the abstract syntax as well as the execution semantics of our state machine dialect. The semantics are of special interest as the application of a different execution semantics will result in a different interpretation of the described dependencies [ACH+13]. For instance, the event visibility which we restrict to one step (cf. Sect. 3.1) has an impact on the dependency analysis. By alleviating the visibility to more than one step, the dependency analysis would identify additional control dependencies as, e.g., two transitions can still synchronize after a sequence of permitted intermediate state machine steps.

Static Backward Slicing. Based on a dependency graph DG_{sm} as well as the original state machine *sm*, we are able to compute a state machine slice for a given slicing criterion *sc*. The algorithm for the backward slice computation is shown in pseudo code in Alg. 4.1, whereas the pseudo codes for the auxiliary functions `initSlice`, `checkDependency`, `checkReachability`, and `wellformSlice` are depicted in Alg. 4.2, Alg. 4.3, Alg. 4.4, and Alg. 4.5, respectively.

Algorithm 4.1.: Backward State Machine Slicing

Input: Slicing Criterion *sc*, State Machine *sm*, and Dependency Graph DG_{sm}
Output: Slice $slice_{sc}^{sm}$

```
1  Function backSlice
2      slice_sc^sm := initSlice(sm, sc);
3      Elem_Next := {sc};
4      while Elem_Next ≠ ∅ do
5          elem ∈ Elem_Next;
6          Elem_Next := Elem_Next \ {elem};
7          slice_sc^sm := checkDependency(elem, sm, slice_sc^sm, DG_sm, Elem_Next);
8          slice_sc^sm := checkReachability(elem, sm, slice_sc^sm, Elem_Next);
9      slice_sc^sm := wellformSlice(sm, slice_sc^sm);
10 return slice_sc^sm;
```

We start the process of backward slicing with the initialization of the empty slice $slice_{sc}^{sm}$ by adding the slicing criterion *sc* (cf. Line 2). Therefore, as shown in the respective Alg. 4.2, we copy the original state machine *sm* and clear the hierarchy functions $\psi_{slice_{sc}^{sm}}$ and $\chi_{slice_{sc}^{sm}}$ as first steps. The hierarchy

Algorithm 4.2.: Function `initSlice`

Input: State Machine *sm* and Slicing Criterion *sc*
Output: Initialized Slice $slice_{sc}^{sm}$

```
1  Function initSlice
2      slice_sc^sm := sm;
3      clear(ψ_slice_sc^sm);
4      clear(χ_slice_sc^sm);
5      forall r ∈ R_slice_sc^sm do
6          r := ({s_0^r}, s_0^r, E_r, L_r, ∅);
7      r_sc := getRegion(sm, slice_sc^sm, sc);
8      if typeOf(sc) == STATE then
9          S_r_sc := S_r_sc ∪ {sc};
10     else
11         T_r_sc := T_r_sc ∪ {sc};
12 return slice_sc^sm;
```

functions will be correctly updated at the end of the slicing computation. As next step, we clear all regions $r \in R_{slice_{sc}^{sm}}$ such that the regions contain solely their initial state and no transitions. At the end of the function `initSlice`, we add the slicing criterion *sc* to its respective region r_{sc} either to the set of states or transitions depending on the element type of *sc*.

After the initialization of slice $slice_{sc}^{sm}$, we initialize the set $Elem_{Next}$ by adding the slicing criterion *sc* (cf. Line 3 in Alg. 4.1). As slicing is a fix-point computation [ACH+13], we use $Elem_{Next}$ to determine if a fix-point is reached, i.e., no new element is added to the slice which is represented by the emptiness of $Elem_{Next}$. Hence, we repeat the following steps until $Elem_{Next}$ is empty (cf. Lines 4 to 8):

1. We select and remove an element *elem* from $Elem_{Next}$ which was added to the slice in previous iterations.
2. For *elem*, we determine those state machine elements it is dependent from by checking the dependency information captured in the dependency graph (cf. Line 7). If we identify new elements, i.e., elements that are not already contained in the slice, we integrate them into the slice and further add them to the set $Elem_{Next}$ for the next iterations. Therefore, as shown in Alg. 4.3, we first get the respective dependency graph node n_{elem} for the state machine element *elem* under consideration. By iterating over all dependencies captured in the dependency graph DG_{sm}, we identify those dependencies, where n_{elem} is the source node, i.e., the target node of the dependency represents the state machine element *elem'* the current element *elem* is dependent from. In case *elem'* is a state, we have to determine whether it is an initial state or not. As the initial states are already part of the slice due to the slice initialization, we solely add *elem'* to the set $Elem_{Next}$ if it is an initial state. Otherwise, we further check whether the state is already contained in the slice. If not, we add the state *elem'* to the slice and also to the set $Elem_{Next}$. In case *elem'* is a transition, we determine whether it is already part of the slice and if not, we add *elem'* to the slice and to $Elem_{Next}$.
3. For *elem*, we further determine state machine elements to be integrated in the slice to ensure the reachability and, therefore, well-formedness of the slice (cf. Line 8). The respective pseudo

Algorithm 4.3.: Function checkDependency

Input: State Machine Element *elem*, State Machine *sm*, Intermediate State Machine Slice $slice_{sc}^{sm}$, Dependency Graph DG_{sm}, and Element Set $Elem_{Next}$
Output: Updated Slice $slice_{sc}^{sm}$

```
1  Function checkDependency
2      n_elem := getNode(N_DG_sm, elem);
3      forall dep ∈ Dep_DG_sm do
4          if dep == (n_elem, l_dep, n_elem') then
5              elem' := getElement(sm, n_elem');
6              r := getRegion(sm, slice_sc^sm, elem');
7              if typeOf(elem') == STATE ∧ !isInitState(r, elem') ∧ elem' ∉ S_r then
8                  S_r := S_r ∪ {elem'};
9                  Elem_Next := Elem_Next ∪ {elem'};
10             else if typeOf(elem') == STATE ∧ isInitState(r, elem') then
11                 Elem_Next := Elem_Next ∪ {elem'};
12             else if typeOf(elem') == TRANSITION ∧ elem' ∉ T_r then
13                 T_r := T_r ∪ {elem'};
14                 Elem_Next := Elem_Next ∪ {elem'};
15 return slice_sc^sm;
```

code is shown in Alg. 4.4. In case *elem* is a transition, we add its source and target states if they are not already contained in the slice. In contrast, if *elem* is a state, we add all its incoming transitions as they enable the reachability of the state in the resulting slice. Again, we add all newly integrated elements also in the set $Elem_{Next}$ to allow for their incorporation in the next iterations.

During the slice computation, we solely focus on the addition of states and transitions to the slice and abstract from their parent regions as they do not provide any additional information usable for the computation. Therefore, as last step of the slice computation, we ensure the well-formedness of the slice via the function `wellformSlice`. As shown in the respective Alg. 4.5, we remove all regions which are empty, i.e., they solely contain their initial state. For all regions which remain in the slice, we update the set of events and labels based on their set of transitions which was determined during the slice computation. Afterwards, the hierarchy functions $\psi_{slice_{sc}^{sm}}$ and $\chi_{slice_{sc}^{sm}}$ are updated, where the set of regions of the slice are incorporated. As last step, the event set of the slice $slice_{sc}^{sm}$ is adapted also based on the set of regions. In the end, we obtain a state machine slice representing a reduced, yet well-formed state machine (cf. Def. 3.8), e.g., the state machine is connected and every state contained in the slice is reachable via a path starting in the initial state of the root region, etc. We refer to Androutsopoulos et al. [ACH+13] for a survey on existing model slicing techniques and control dependencies.

The list of symbols used for the definition of state machine slicing is summarized in Tab. 4.1. To recapitulate, state machine slicing reduces a state machine *sm* by abstracting from state machine elements that do not influence a given slicing criterion *sc*, e.g., a transition. The result is called a state machine slice $slice_{sc}^{sm}$ and represents a reduced, yet well-formed state machine. For the computation, the execution dependencies between state machine elements have to be determined and

Algorithm 4.4.: Function `checkReachability`

Input: State Machine Element *elem*, State Machine *sm*, Intermediate State Machine Slice $slice_{sc}^{sm}$, and Element Set $Elem_{Next}$
Output: Updated Slice $slice_{sc}^{sm}$

Function *checkReachability*
  $r_{sm} := getRegion(sm, elem)$;
  $r_{slice_{sc}^{sm}} := getRegion(sm, slice_{sc}^{sm}, elem)$;
  if $typeOf(elem) == STATE$ **then**
    forall $t \in T_{r_{sm}}$ **do**
      if $hasTargetState(t, elem)$ **then**
        if $t \notin T_{r_{slice_{sc}^{sm}}}$ **then**
          $T_{r_{slice_{sc}^{sm}}} := T_{r_{slice_{sc}^{sm}}} \cup \{t\}$;
          $Elem_{Next} := Elem_{Next} \cup \{t\}$;
  else
    $s_{source} := getSourceState(elem)$;
    $s_{target} := getTargetState(elem)$;
    if $s_{target} \notin S_{r_{slice_{sc}^{sm}}}$ **then**
      $S_{r_{slice_{sc}^{sm}}} := S_{r_{slice_{sc}^{sm}}} \cup \{s_{target}\}$;
    if $s_{source} \notin S_{r_{slice_{sc}^{sm}}}$ **then**
      $S_{r_{slice_{sc}^{sm}}} := S_{r_{slice_{sc}^{sm}}} \cup \{s_{source}\}$;
      $Elem_{Next} := Elem_{Next} \cup \{s_{source}\}$;
return $slice_{sc}^{sm}$;

Algorithm 4.5.: Function `wellformSlice`

Input: State Machine *sm* and Intermediate State Machine Slice $slice_{sc}^{sm}$
Output: Slice $slice_{sc}^{sm}$

Function *wellformSlice*
  forall $r \in R_{slice_{sc}^{sm}}$ **do**
    if $S_r == \{s_0^r\} \wedge r \neq r_0^{slice_{sc}^{sm}}$ **then**
      $R_{slice_{sc}^{sm}} := R_{slice_{sc}^{sm}} \setminus \{r\}$;
    else
      $E_r := updateEvents(T_r)$;
      $L_r := updateLabels(T_r)$;
  forall $s \in S_{slice_{sc}^{sm}}$ **do**
    $\psi_{slice_{sc}^{sm}}(s) = \psi_{sm}(s) \cap R_{slice_{sc}^{sm}}$;
  forall $r \in R_{slice_{sc}^{sm}} \setminus \{r_0^{slice_{sc}^{sm}}\}$ **do**
    $\chi_{slice_{sc}^{sm}}(r) = \chi_{sm}(r)$;
  $E_{slice_{sc}^{sm}} := updateEvents(R_{slice_{sc}^{sm}})$;
return $slice_{sc}^{sm}$;

Table 4.1: Symbol Summary of State Machine Slice Definition

Symbol	Description
sc	Slicing criterion
$slice_{sc}^{sm}$	State machine slice of state machine sm w.r.t. slicing criterion sc
DG_{sm}	Dependency graph for state machine sm
$n; N_{sm}; \mathcal{N}$	State machine element node; Finite set of element nodes; Universe of element nodes
Dep_{sm}	Dependency edge relation
$\mathcal{L}_{Dep}$	Finite set of dependency labels
sd	Synchronization dependency
tcd	Transition control dependency
gcd	Global control dependency
rcd	Refinement control dependency

captured in a respective dependency graph DG_{sm}. The graph comprises for each element of state machine sm, i.e., states and transitions, an element node $n \in N_{sm}$. If a dependency between two state machine elements exists, a respective dependency edge is captured in Dep_{sm}. In this thesis, we apply the control dependencies $\mathcal{L}_{Dep} = (sd, tcd, gcd, rcd)$ defined by Kamischke et al. [KLB12], namely synchronization dependency sd, transition control dependency tcd, global control dependency gcd, and refinement control dependency rcd.

Example 4.1: Backward Model Slicing

Consider the sample core state machine from Ex. 3.1 depicted in Fig. 3.1 again. The respective dependency graph $DG_{sm_{v_{core}}} = (N_{sm_{v_{core}}}, Dep_{sm_{v_{core}}})$ is shown in Fig. 4.1a and is defined by

- $N_{sm_{v_{core}}} = \{s_0, s_1, a_1, a_2, b_1, b_2, b_3, c_1, c_2, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$, and
- $Dep_{sm_{v_{core}}} = \{(a_1, rcd, s_0), (b_1, rcd, s_0), (t_1, gcd, a_1), (t_2, tcd, t_1), (t_6, tcd, t_1), (t_6, tcd, t_5), (t_5, gcd, b_2), (t_3, gcd, b_1), (t_4, gcd, b_2), (t_7, gcd, c_1), (c_1, rcd, s_1), (t_8, gcd, c_2)\}$.

For instance, transitions t_2 and t_1 from region `A` are transition control dependent as t_1 generates the event e_2^τ via its event broadcast when the transition is taken and t_2 requires e_2^τ as triggering event and, therefore, consumes the event in the next step.

In Fig. 4.1b, the backward slice for the slicing criterion transition t_2 is depicted, where those parts of the state machine which are not contained in the slice are colored in gray. The slice and, hence, the reduced, yet well-formed state machine $slice_{t_2}^{sm_{v_{core}}} = (R, r_0, \psi, E)$ is defined by

- $R = \{\texttt{Root}, \texttt{A}\}$ with $\texttt{Root} = (S, s_0, T, E, L) = (\{s_0\}, s_0, \emptyset, \emptyset, \emptyset)$, and $\texttt{A} = (\{a_1, a_2\}, a_1, \{t_1, t_2\}, \{e_1^I, e_2^\tau, e_3^\tau, e_4^O\}, \{(e_1^I / \{e_2^\tau, e_4^O\}), (e_2^\tau / \{e_3^\tau\})\})$
- $r_0 = \texttt{Root}$,
- $\psi : \psi(s_0) = \{A\}$, and
- $E = \{e_1^I\} \cup \{e_2^\tau, e_3^\tau\} \cup \{e_4^O\}$.

The slice computation (cf. Alg. 4.1) starts with the transition t_2, and first check for its dependencies. As transition t_2 is transition control dependent to transition t_1, we add t_1 to the slice. In addition, we add the source and target state of t_2 based on the reachability check.

The transition t_1 as well as the states a_1 and a_2 are also added to the set $Elem_{Next}$. In the next iteration, we select t_1 and check for dependencies as well as for reachability. Based on the global control dependency between t_1 and a_1, we would add the state to the slice if it was not already contained. The same holds for state a_2 which would be added to ensure reachability. For the next iterations, we first focus on state a_1 and then a_2, where we solely add state s_0 to the slice based on the refinement control dependency between a_1 and s_0. The fix-point is reached, i.e., the slice $slice_{t_2}^{sm_{v_{core}}}$ is computed after checking the last added element s_0 for dependencies and reachability.

In Fig. 4.1c, the backward slice $slice_{t_5}^{sm_{v_{core}}}$ for transition t_5 is depicted. Compared to $slice_{t_2}^{sm_{v_{core}}}$, the slice $slice_{t_5}^{sm_{v_{core}}}$ comprises solely the region D and abstract from the remaining behavior specified in the core state machine $sm_{v_{core}}$.

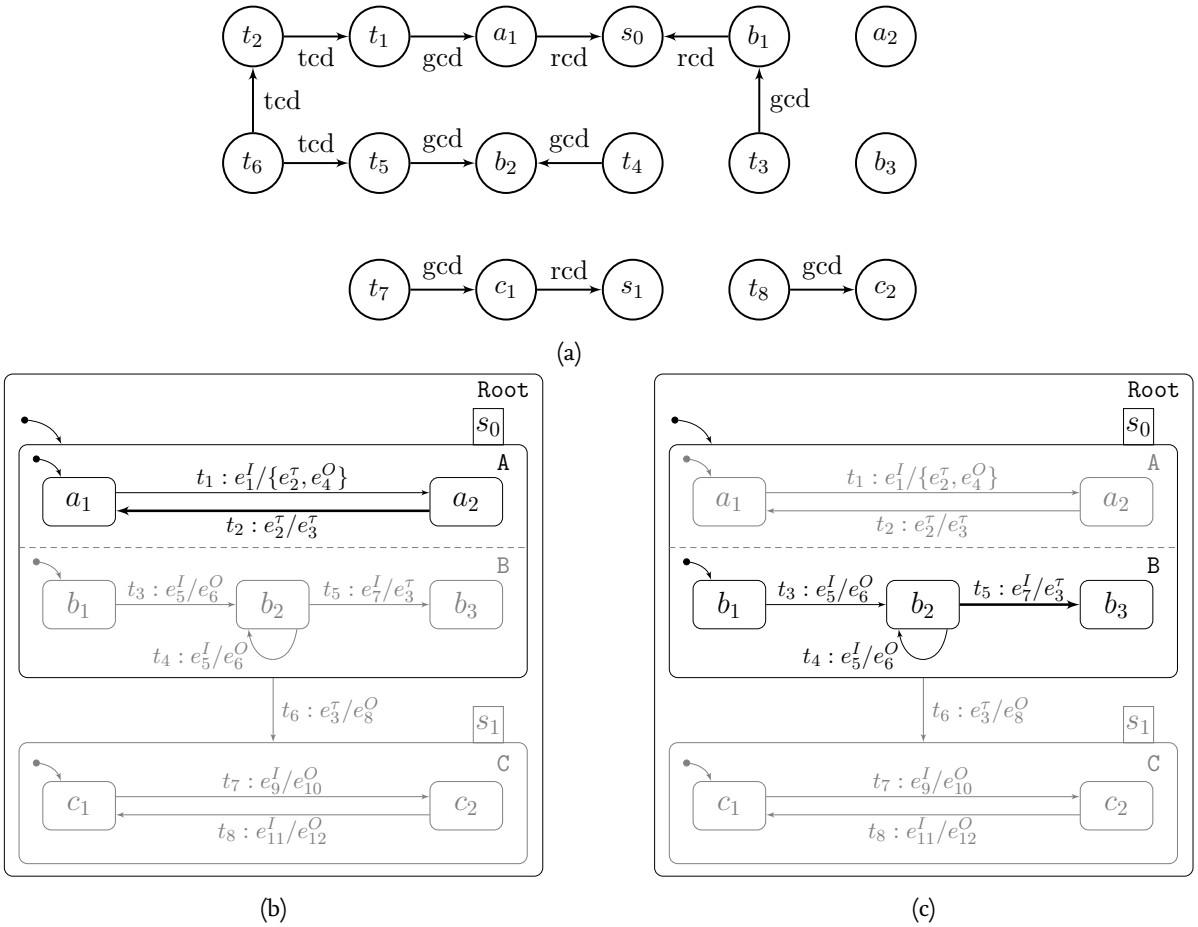


Figure 4.1: Sample Dependency Graph $DG_{sm_{v_{core}}}$ for Core State Machine $sm_{v_{core}}$ (a), Sample State Machine Slice $slice_{t_2}^{sm_{v_{core}}}$ for Transition t_2 (b), and Sample State Machine Slice $slice_{t_5}^{sm_{v_{core}}}$ for Transition t_5 (c)

4.1.2 Incremental Model Slicing

In the context of SPLs, applying model slicing individually for each variant $v \in \mathbb{V}$ of an SPL is, in general, infeasible. The vast number of potential variants and the shared commonality lead to an

inefficient analysis if the generation of the dependency graph and the slice computations are performed naively for every variant anew. By taking the commonality and variability between variants during analysis into account [TAK+14], model slicing will facilitate an efficient SPL analysis.

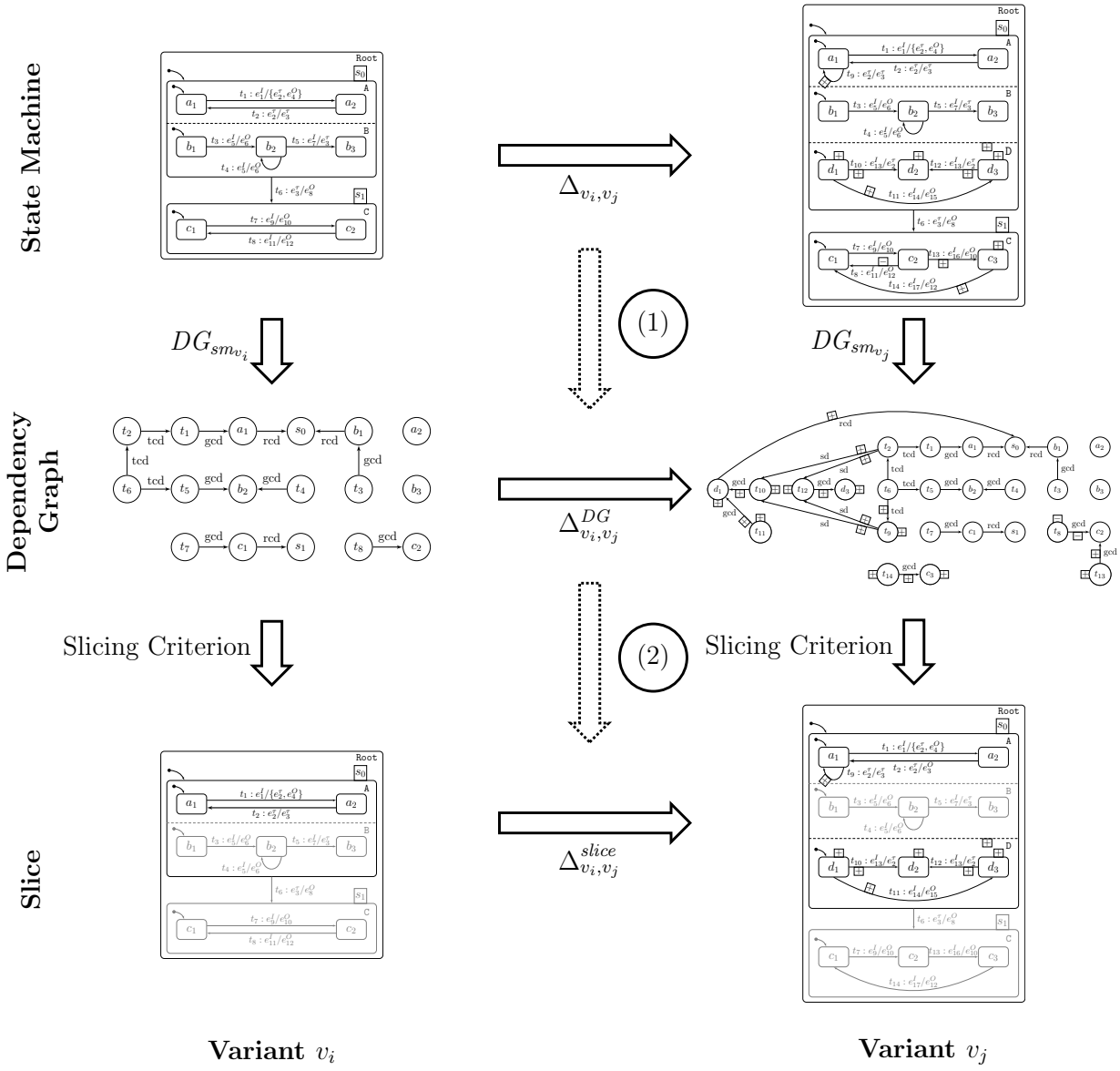


Figure 4.2: Overview Incremental Model Slicing

As already discussed in Chapt. 3, delta modeling allows for the explicit specification of the commonality and differences between arbitrary variants by means of (regression) deltas [Sch10; CHS15; LLL+14]. By adopting the concept of delta modeling for model slicing, we define an incremental slicing technique, where we exploit the commonality and focus on the differences such that the dependency graph and slices are not computed completely anew for each variant. Furthermore, the focus on the differences facilitates, e.g., the reasoning about the change impact between two consecutively tested variants. In Fig. 4.2, we provide an overview of incremental model slicing. On the left hand side, the standard slicing procedure as described above is shown. For a state machine sm_{v_i}

of variant v_i, we first generate the dependency graph $DG_{sm_{v_i}}$. Afterwards, we use the dependency graph, the state machine and a slicing criterion *sc* to compute a slice representing the partial behavior of the variant that influences the criterion during execution. In contrast to apply this procedure anew for the variant v_j on the right hand side, we exploit the differences between the state machines captured in a regression delta Δ_{v_i,v_j}. We use the change operations representing the differences between the state machines to derive respective changes for the dependency graph (cf. Fig. 4.2 (1)), e.g., the addition of an element node due to the addition of its state machine element. We capture the differences between the dependency graphs in a *dependency graph regression delta* $\Delta^{DG}_{v_i,v_j}$ and exploit it to recompute the slice for the slicing criterion *sc* (cf. Fig. 4.2 (2)). During the recomputation, we capture the differences between the slices in a *slice regression delta* $\Delta^{slice_{sc}}_{v_i,v_j}$. The change operations captured in a slice regression delta indicate that the execution dependencies of the state machine element that is used as slicing criterion have changed. This information facilitates the guidance of the retest test selection in our model-based regression testing framework (cf. Chapt. 5).

In the following, we first describe the incremental adaptation of variant-specific dependency graphs (cf. Fig.4.2, (1)), and afterwards, we explain the incremental slice computation (cf. Fig.4.2, (2)). Furthermore, we outline how to apply incremental model slicing for change impact analysis.

Incremental Dependency Graph Adaptation. When stepping from a variant $v_i \in \mathbb{V}$ to a subsequent variant $v_j \in \mathbb{V}$ to be analyzed, we use the state machine regression delta Δ_{v_i,v_j} as starting point for the incremental adaptation of the dependency graph $DG_{sm_{v_i}}$ to obtain $DG_{sm_{v_j}}$. For the adaptation, we first add and remove element nodes from the dependency graph w.r.t. change operations captured in the regression delta. Afterwards, we solely focus on the dependency analysis, where we check whether (1) dependency edges are still valid and remain in the graph, (2) are obsolete and have to be removed, or (3) are new and have to be added to the graph. Therefore, we derive respective change operations to transform $DG_{sm_{v_i}}$ into $DG_{sm_{v_j}}$. A *dependency graph change operation* specifies (1) the addition and removal of element nodes, and (2) the addition and removal of dependency edges.

Definition 4.3: Dependency Graph Change Operation

Let $\mathcal{OP}^{DG}$ be the universe of all dependency graph change operations defined over the universe $\mathcal{N}$ of all element nodes and the dependency edge relation $\mathcal{N} \times \mathcal{L}_{Dep} \times \mathcal{N}$. The universe $\mathcal{N}$ is further defined by the universe of all states $\mathcal{S}$ and all transitions $\mathcal{T}$. A *dependency graph change operation* $op^{DG} \in \mathcal{OP}^{DG}$ defines one of the following transformations:

- add n, i.e., an element node $n \in \mathcal{N}$ is added,
- rem n, i.e., an element $n \in \mathcal{N}$ is removed,
- add (n, l_{dep}, n'), i.e., a dependency edge $(n, l_{dep}, n') \in \mathcal{N} \times \mathcal{L}_{Dep} \times \mathcal{N}$ is added, and
- rem (n, l_{dep}, n'), i.e., a dependency edge $(n, l_{dep}, n') \in \mathcal{N} \times \mathcal{L}_{Dep} \times \mathcal{N}$ is removed.

We capture the derived change operations in a dependency graph regression delta which is used afterwards in the step of incremental slice computation.

Definition 4.4: Dependency Graph Regression Delta

Let apply_{DG} be the incremental application function to transform a dependency graph DG_{sm} into another dependency graph $DG_{sm'}$ based on a given set of dependency graph change op-

erations. A *dependency graph regression delta* $\Delta^{DG}_{v_i,v_j} = \{op^{DG}_1, \ldots, op^{DG}_m\}$ captures all dependency graph change operations such that $DG_{sm_{v_j}} = \mathsf{apply}_{DG}(DG_{sm_{v_i}}, \Delta^{DG}_{v_i,v_j})$ holds.

The incremental application function apply_{DG} is defined similar to apply_{δ} for the incremental state machine delta application (cf. Def. 3.12) as well as $\mathsf{apply}_{\delta^H}$ for the higher-order delta application (cf. Def. 3.17). Based on those definitions, we propose the incremental dependency graph adaptation as follows, where the respective algorithm is shown in Alg. 4.6 in pseudo code [LBS15].

Algorithm 4.6.: Incremental Dependency Graph Adaptation [LBS15]

Input: Dependency Graph $DG_{sm_{v_i}}$, State Machine sm_{v_j}, and State Machine Regression Delta Δ_{v_i,v_j}
Output: Dependency Graph $DG_{sm_{v_j}}$ and Dependency Graph Regression Delta $\Delta^{DG}_{v_i,v_j}$

```
1  Function incDepGraphAdapt
2      Δ^DG_{v_i,v_j} := initDGRegDelta(Δ_{v_i,v_j}, DG_{sm_{v_i}});
3      DG_{sm_{v_j}} := apply_DG(DG_{sm_{v_i}}, Δ^DG_{v_i,v_j});
4      N_Next := ∅;
5      N_Next := updateNext(Δ^DG_{v_i,v_j}, N_Next);
6      while N_Next ≠ ∅ do
7          n ∈ N_Next;
8          N_Next := N_Next \ {n};
9          foreach n' ∈ N_{sm_{v_j}} do
10             Δ^n_DG := checkDependencies(n, n', sm_{v_j}, DG_{sm_{v_j}}, Δ^n_DG);
11         N_Next := updateNext(Δ^n_DG, N_Next);
12         DG_{sm_{v_j}} := apply_DG(DG_{sm_{v_j}}, Δ^n_DG);
13         Δ^DG_{v_i,v_j} := Δ^DG_{v_i,v_j} ∪ Δ^n_DG;
14 return DG_{sm_{v_j}}, Δ^DG_{v_i,v_j};
```

For the adaptation, we require as input (1) the dependency Graph $DG_{sm_{v_i}}$ of the previous variant v_i, (2) the state machine sm_{v_j} of the current variant v_j to be analyzed, and (3) the state machine regression delta Δ_{v_i,v_j} capturing the differences between both variants. The algorithm provides as result the adapted dependency graph $DG_{sm_{v_j}}$ for the variant v_j as well as the dependency graph regression delta $\Delta^{DG}_{v_i,v_j}$. As first step of the adaptation (cf. Line 2), we analyze the state machine regression delta Δ_{v_i,v_j} and initialize the dependency graph regression delta $\Delta^{DG}_{v_i,v_j}$. The pseudo code of the function `initDGRegDelta` is shown in Alg. 4.7. By iterating over all state machine change operations captured in the regression delta Δ_{v_i,v_j}, we derive for the addition and removal of states as well as transitions respective dependency graph change operations and add them to the dependency graph regression delta $\Delta^{DG}_{v_i,v_j}$ to be initialized. In case of a removal of a state machine element, we also derive removals of dependency edges which have a node to be removed as source or target node. Furthermore, we cope with the modification of a transition by removing the original transition node and adding the modified transition as new element node. This remove-add encoding facilitates the detection of the influence of the modification also during the incremental slicing process. Based on the application of the control dependencies, we do not have to remove further

Algorithm 4.7.: Function `initDGRegDelta`

Input: State Machine Regression Delta Δ_{v_i,v_j} and Previous Dependency Graph $DG_{sm_{v_i}}$
Output: Initialized Dependency Graph Regression Delta $\Delta^{DG}_{v_i,v_j}$

Function *initDGRegDelta*
  forall $op \in \Delta_{v_i,v_j}$ **do**
    if $op == \mathsf{add}\, s$ **then**
      $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{add}\, n_s\}$;
    else if $op == \mathsf{rem}\, s$ **then**
      $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{rem}\, n_s\}$;
      forall $dep \in DG_{sm_{v_i}}$ **do**
        if $dep == (n_s, l_{dep}, n') \vee dep == (n', l_{dep}, n_s)$ **then**
          $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{rem}\, dep\}$;
    else if $op == \mathsf{add}\, t$ **then**
      $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{add}\, n_t\}$;
    else if $op == \mathsf{rem}\, t$ **then**
      $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{rem}\, n_t\}$;
      forall $dep \in DG_{sm_{v_i}}$ **do**
        if $dep == (n_t, l_{dep}, n') \vee dep == (n', l_{dep}, n_t)$ **then**
          $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{rem}\, dep\}$;
    else if $op == \mathsf{mod}(t, l')$ **then**
      $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{rem}\, n_t\}$;
      forall $dep \in DG_{sm_{v_i}}$ **do**
        if $dep == (n_t, l_{dep}, n') \vee dep == (n', l_{dep}, n_t)$ **then**
          $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{rem}\, dep\}$;
      $\Delta^{DG}_{v_i,v_j} := \Delta^{DG}_{v_i,v_j} \cup \{\mathsf{add}\, n_{t'}\}$;
return $\Delta^{DG}_{v_i,v_j}$;

Algorithm 4.8.: Function `updateNext`

Input: Set of Dependency Graph Change Operations Δ^{DG} and Set of Element Nodes N_{Next}
Output: Updated Set of Element Nodes N_{Next}

Function *updateNext*
  forall $op^{DG} \in \Delta^{DG}$ **do**
    if $op^{DG} == \mathsf{add}\, n$ **then**
      $N_{Next} := N_{Next} \cup \{n\}$;
    else if $op^{DG} == \mathsf{add}\, (n, l, n')$ **then**
      $N_{Next} := N_{Next} \cup \{n, n'\}$;
    else if $op^{DG} == \mathsf{rem}\, (n, l, n')$ **then**
      if $\mathsf{rem}\, n \notin \Delta^{DG}$ **then**
        $N_{Next} := N_{Next} \cup \{n\}$;
      if $\mathsf{rem}\, n' \notin \Delta^{DG}$ **then**
        $N_{Next} := N_{Next} \cup \{n'\}$;
return N_{Next};

Algorithm 4.9.: Function `checkDependencies`

Input: Dependency Graph Node n, Dependency Graph Node n', State machine sm_{v_j}, Dependency Graph $DG_{sm_{v_j}}$, and Set of Dependency Graph Change Operations Δ^n_{DG}
Output: Updated Set of Dependency Graph Change Operations Δ^n_{DG}

1 **Function** *checkDependencies*
2 $elem_n := getElement(sm_{v_j}, n)$;
3 $elem_{n'} := getElement(sm_{v_j}, n')$;
4 **if** $(n, l, n') \notin Dep_{DG_{sm_{v_j}}} \wedge areDependent(elem_n, elem_{n'}, sm_{v_j})$ **then**
5 $\Delta^n_{DG} := \Delta^n_{DG} \cup \{\mathsf{add}\,(n, l_{dep}, n')\}$;
6 **else if** $(n', l, n) \notin Dep_{DG_{sm_{v_j}}} \wedge areDependent(elem_{n'}, elem_n, sm_{v_j})$ **then**
7 $\Delta^n_{DG} := \Delta^n_{DG} \cup \{\mathsf{add}\,(n', l_{dep}, n)\}$;
8 **return** Δ^n_{DG};

dependency edges. Those control dependencies have no side-effects on each other and, therefore, the remaining dependencies captured in the dependency graph are still valid.

As next step (cf. Line 3), we apply the initialized dependency graph regression delta $\Delta^{DG}_{v_i,v_j}$ on the dependency graph $DG_{sm_{v_i}}$ of the previous analyzed variant v_i to obtain the intermediate dependency graph $DG_{sm_{v_j}}$ of variant v_j to be analyzed comprising only valid nodes w.r.t. state machine elements in sm_{v_j} and no obsolete dependencies. Furthermore, we collect the nodes which are added via the dependency graph regression delta as well as nodes which are connected to removed dependency edges in the set N_{Next} (cf. Line 5) as shown in Alg. 4.8. For newly added nodes, we have not yet applied the dependency analysis, whereas for those nodes connected to removed dependencies, we have to determine whether new dependencies are introduced.

After this initialization phase, we start to analyze for new dependencies by iterating over the set N_{Next} (cf. Line 6 to 13). The set N_{Next} solely contains (1) nodes which are newly added to the dependency graph, i.e., for those nodes, the dependency analysis was not yet applied, and (2) existing nodes which are influenced by changes to the dependency graph, i.e., a connecting dependency edge was added or removed during the incremental adaptation. In contrast, for existing nodes which are not influenced by dependency graph changes, we do not have to apply the dependency analysis again. We select and remove an element node in each iteration and check for this node against all other element nodes of the dependency graph whether their represented state machine elements are dependent on each other (cf. Line 10). The pseudo code of the function `checkDependencies` is shown in Alg. 4.9. For the determination if two state machine elements are dependent, we refer to the definition of the incorporated control dependencies in Sect. 4.1.1. If a not yet captured dependency is detected, we extend the set Δ^n_{DG} of dependency graph change operations for the current node n by a respective addition of the dependency edge. To prevent a redundant analysis, we record the already analyzed element node pairs. We omit the recording from the algorithm to make the presentation more graspable. Based on the determined set Δ^n_{DG}, we update the set N_{Next} such that we add those element nodes to which the new dependency edges are connected (cf. Line 11 and Alg. 4.8). Similar to the recorded element pairs, we record whether an element node was already contained in N_{Next} and skip its addition in such case. As last steps of the iteration (cf. Line 12 and 13), we apply the

set Δ^{n}_{DG} to the intermediate dependency graph $DG_{sm_{v_j}}$ and extend the dependency graph regression delta $\Delta^{DG}_{v_i,v_j}$.

During the adaptation, we solely have to check and adapt parts of the dependency graph which are affected by the changes made to the respective variant-specific state machine, whereas common parts of the dependency graphs between subsequent variants to be analyzed are preserved. Thus, our incremental dependency graph adaptation is strongly dependent on the differences between subsequent variants captured as change operations in a state machine regression delta. In this context, the number of change operations is not necessarily the dominant factor, but rather their resulting impact on dependent behavior. Local changes have a lower impact on the adaptation of the dependency graph such that we achieve a reduction of the dependency analysis effort by exploiting the commonality between variant-specific state machines. In contrast, distributed changes have a strong impact on the dependency analysis and, therefore, on the dependency graph adaptation yielding, e.g., a new dependency graph without effort reduction in the *worst case*.

The incremental adaptation of the dependency graph terminates since (1) the set $N_{sm_{v_j}}$ of element nodes captured in the graph $DG_{sm_{v_j}}$ is finite, and (2) the set N_{Next} is solely extended by element nodes which are not already checked. As result of the adaptation, we obtain the valid dependency graph $DG_{sm_{v_j}}$ of the current variant v_j under analysis as well as the differences to the dependency graph $DG_{sm_{v_j}}$ of the previously analyzed variant v_i recorded in a dependency graph regression delta $\Delta^{DG}_{v_i,v_j}$.

Example 4.2: Dependency Graph Adaptation

Consider the dependency graph $DG_{sm_{v_{core}}}$ from Ex. 4.1 shown in Fig. 4.1a capturing the result of the dependency analysis for the core state machine $sm_{v_{core}}$. By stepping from the core variant v_{core} to variant v_1, we apply the state machine regression delta $\Delta_{v_{core},v_1} = \Delta_{v_1}$ from Ex. 3.3 to transform the core state machine $sm_{v_{core}}$ into the state machine sm_{v_1} of variant v_1. The resulting dependency graph $DG_{sm_{v_1}}$ is depicted in Fig. 4.3, where added and removed graph elements are marked with either a $+$ or a $-$, respectively. First, the element nodes for the newly added state machine elements d_1, d_2, d_3, c_3, t_{10}, t_{11}, t_{12}, t_{13}, and t_{14} are added to the dependency graph as well as to the set N_{Next}. Second, the element node for the removed transition t_8 is removed and further its connected dependency edge (t_8, gcd, c_2). Hence, the node c_2 is also added to N_{Next}. Third, we iterate over the set N_{Next} to determine and add the missing dependencies (t_{10}, gcd, d_1), (t_{11}, gcd, d_1), (t_2, sd, t_{10}), (t_2, sd, t_{12}), (t_9, sd, t_{10}), (t_9, sd, t_{12}), (t_{12}, gcd, d_3), (d_a, rcd, s_0), (t_6, tcd, t_9), (t_{14}, gcd, c_3), and (t_{13}, gcd, c_2). Based on the new dependency edges (d_a, rcd, s_0) as well as (t_6, tcd, t_9), we add the element nodes s_0 and t_6 to the set N_{Next}. Therefore, we also recheck for both nodes whether new dependencies have to be added to the dependency graph. As we can see, most of the dependency graph stays unchanged reducing the effort for dependency analysis compared to generating the dependency graph completely anew. In the end, we capture the described dependency graph change operations also marked in Fig. 4.3 in the respective dependency graph regression delta $\Delta^{DG}_{v_{core},v_1}$ to be used for the incremental slice computation.

Incremental Slice Computation. The incremental slice computation is performed similar to the dependency graph adaptation by incorporating the changes captured in the determined dependency

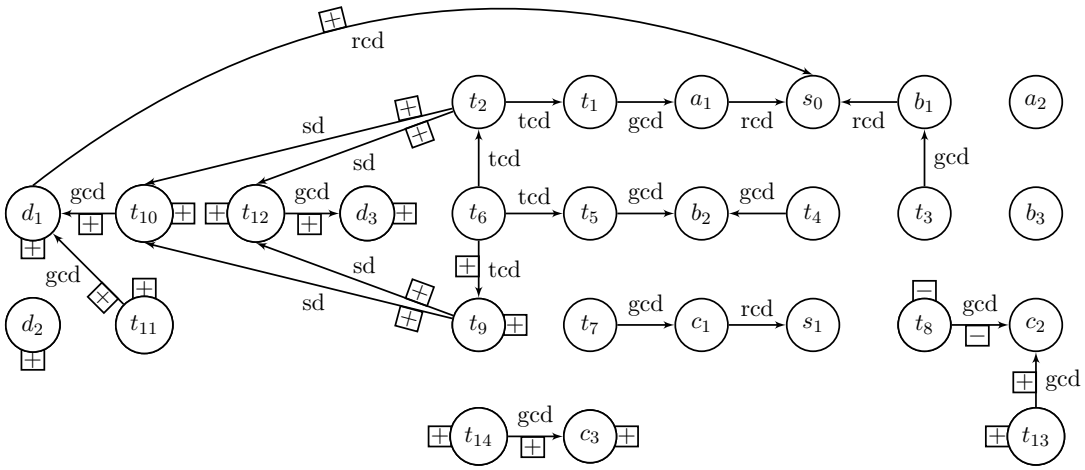


Figure 4.3: Dependency Graph $DG_{sm_{v_1}}$ for State Machine sm_{v_1} Including Graph Differences to Dependency Graph $DG_{sm_{v_{core}}}$

graph regression delta $\Delta^{DG}_{v_i,v_j}$. We compute a new slice $slice_{sc}^{sm_{v_j}}$ for a given slicing criterion sc based on an existing slice $slice_{sc}^{sm_{v_i}}$ for the same criterion already computed for the previously analyzed variant v_i. In contrast to the dependency graph adaptation, we do not achieve a reduction of the computation effort as we perform a recomputation of the slice. However, the incrementallity and, therefore, the benefit of this part of our incremental slicing technique is the direct identification of differences between the new and the previous slice during the slice computation saving the effort for comparing and diffing the slices. Those differences indicate the impact of the changes to the variant-specific state machine sm_{v_j} to the potential execution behavior of the given slicing criterion sc, e.g., a transition, as dependencies to other state machine elements are newly introduced or removed affecting the execution of sc.

To capture the slice differences, we also adopt the concept of delta modeling [Sch10; CHS15] for the slice computation. Therefore, we define change operations similar as for the incremental dependency graph adaptation to transform the previous slice $slice_{sc}^{sm_{v_i}}$ w.r.t. a slicing criterion sc of an already analyzed variant v_i into the new slice $slice_{sc}^{sm_{v_j}}$ for variant v_j to be analyzed. A *slice change operation* specifies the addition and removal of (1) states and (2) transitions.

Definition 4.5: Slice Change Operation

Let $\mathcal{OP}^{Slice} \subset \mathcal{OP}$ be the universe of all slice change operations which is a subset of the universe of all change operations $\mathcal{OP}$. A *slice change operation* $op^{Slice} \in \mathcal{OP}^{Slice}$ defines one of the following transformations:

- add s, i.e., a state $s \in \mathcal{S}$ is added,
- rem s, i.e., a state $s \in \mathcal{S}$ is removed,
- add t, i.e., a transition $t \in \mathcal{T}$ is added, and
- rem t, i.e., a transition $t \in \mathcal{T}$ is removed.

We capture the determined change operations between two slices in a *slice regression delta* representing the result of the incremental analysis, e.g., change impact analysis.

Definition 4.6: Slice Regression Delta

Let apply_{slice} be the incremental slicing application function to transform a slice $slice_{sc}$ into another slice $slice'_{sc}$ based on a given set of slice change operations. A *slice regression delta* $\Delta^{slice_{sc}}_{v_i,v_j} = \{op_1^{Slice}, \ldots, op_m^{Slice}\}$ captures all slice change operations such that $slice_{sc}^{sm_{v_j}} = \mathsf{apply}_{slice}(slice_{sc}^{sm_{v_i}}, \Delta^{slice_{sc}}_{v_i,v_j})$ holds.

The function apply_{slice} is defined similar to the application functions apply_{δ} (cf. Def. 3.12) as well as $\mathsf{apply}_{\delta^H}$ (cf. Def. 3.17) for (higher-order) delta state machines, and apply_{DG} for the dependency graph adaptation. In the following, we describe the process of the incremental slice computation, where the respective algorithm is shown in Alg. 4.10 in pseudo code [LBS15]. The algorithm takes the state machine sm_{v_j} and the dependency graph $DG_{sm_{v_j}}$ of the current variant v_j, a slicing criterion sc, the dependency graph regression delta $\Delta^{DG}_{v_i,v_j}$, and an existing state machine slice $slice_{sc}^{sm_{v_i}}$ w.r.t. sc of the previously analyzed variant v_i as input. As result, we obtain the new slice $slice_{sc}^{sm_{v_j}}$ w.r.t. the slicing criterion sc for the variant v_j under analysis and the slice regression delta $\Delta^{slice_{sc}}_{v_i,v_j}$ capturing the differences to the previous slice $slice_{sc}^{sm_{v_i}}$ of variant v_i. If no previous slice $slice_{sc}^{sm_{v_i}}$ for the given sc exists in the previously analyzed variant v_i, we apply the standard slicing procedure of Alg. 4.1 as described above. The pseudo codes for the auxiliary functions `initSliceRegDelta`, `checkDependency`, and `checkReachability` are shown in Alg. 4.11, Alg. 4.12, and Alg. 4.13, respectively.

We start the incremental slice computation (cf. Line 2) by initializing the slice regression delta based on the change operations captured in the dependency graph regression delta $\Delta^{DG}_{v_i,v_j}$ and the previous slice $slice_{sc}^{sm_{v_i}}$. As shown in Alg. 4.11, we take the removals of element nodes into account as those operations indicate respective changes to the slice in terms of removals of obsolete state machine elements. Hence, we derive initial slice change operations such that state machine elements *elem* contained in slice $slice_{sc}^{sm_{v_i}}$ are removed if corresponding remove operations of their element nodes are defined in the dependency graph regression delta, where the symbol $\dashv$ represents the case that the state machine element to be potentially removed is not contained in the slice.

After the regression delta initialization, we apply the preliminary slice regression delta $\Delta^{slice_{sc}}_{v_i,v_j}$ to the previous slice $slice_{sc}^{sm_{v_i}}$ to obtain the preliminary slice $slice_{sc}^{sm_{v_j}}$ for the current variant v_j under analysis which is used as basis for the remaining process of incremental slice computation. As last step, we initialize the set $Elem_{Next}$ with the slicing criterion sc to start the recomputation of the slice (cf. Line 5 to 11). In contrast to the incremental dependency graph adaptation, where the set N_{Next} indicates for which element the dependency analysis has to be applied, the set $Elem_{Next}$ refers to the elements for which the slice has to be extended and change operations have to be derived.

We start each iteration (cf. Lines 5 to 11) just as the incremental dependency graph adaptation by selecting and removing a state machine element *elem* from the set $Elem_{Next}$. For this element, we first determine dependent elements (cf. Line 8) to be added to the slice by examining the dependency graph as shown in Alg. 4.12. Hence, we check for each existing dependency to another element *elem'* whether *elem'* is already contained in the slice or not. We perform the determination similar to the function `checkDependency` of the standard slicing computation depicted in Alg. 4.3 with the difference that we do not add elements directly, but rather derive respective slice change operations. In case the element *elem'* is contained, we solely extend the set $Elem_{Next}$ by *elem'* to continue the slice

Algorithm 4.10.: Incremental Slice Computation [LBS15]

Input: State Machine sm_{v_j}, Dependency Graph $DG_{sm_{v_j}}$, Slicing Criterion sc, Dependency Graph Regression Delta $\Delta^{DG}_{v_i,v_j}$, and State Machine Slice $slice^{sm_{v_i}}_{sc}$

Output: State Machine Slice $slice^{sm_{v_j}}_{sc}$, and Slice Regression Delta $\Delta^{slice_{sc}}_{v_i,v_j}$

1 **Function** *incBackwardSlice*
2 $\quad \Delta^{slice_{sc}}_{v_i,v_j} := initSliceRegDelta(slice^{sm_{v_i}}_{sc}, \Delta^{DG}_{v_i,v_j})$;
3 $\quad slice^{sm_{v_j}}_{sc} := \mathsf{apply}_{slice}(slice^{sm_{v_i}}_{sc}, \Delta^{slice_{sc}}_{v_i,v_j})$;
4 $\quad Elem_{Next} := \{sc\}$;
5 $\quad$ **while** $Elem_{Next} \neq \emptyset$ **do**
6 $\quad\quad elem \in Elem_{Next}$;
7 $\quad\quad Elem_{Next} := Elem_{Next} \setminus \{elem\}$;
8 $\quad\quad \Delta^{elem}_{slice_{sc}} := checkDependency(elem, sm_{v_j}, slice^{sm_{v_j}}_{sc}, DG_{sm_{v_j}}, Elem_{Next})$;
9 $\quad\quad \Delta^{elem}_{slice_{sc}} := checkReachability(elem, sm_{v_j}, slice^{sm}_{sc}, Elem_{Next}, \Delta^{elem}_{slice_{sc}})$;
10 $\quad\quad slice^{sm_{v_j}}_{sc} := \mathsf{apply}_{slice}(slice^{sm_{v_j}}_{sc}, \Delta^{elem}_{slice_{sc}})$;
11 $\quad\quad \Delta^{slice_{sc}}_{v_i,v_j} := \Delta^{slice_{sc}}_{v_i,v_j} \cup \Delta^{elem}_{slice_{sc}}$;
12 $\quad \Delta^{obsolete}_{slice_{sc}} := remObsoleteElems(Elems_{slice^{sm_{v_i}}_{sc}} \setminus Elems_{slice^{sm_{v_j}}_{sc}})$;
13 $\quad slice^{sm_{v_j}}_{sc} := \mathsf{apply}_{slice}(slice^{sm_{v_j}}_{sc}, \Delta^{obsolete}_{slice_{sc}})$;
14 $\quad \Delta^{slice_{sc}}_{v_i,v_j} := \Delta^{slice_{sc}}_{v_i,v_j} \cup \Delta^{obsolete}_{slice_{sc}}$;
15 $\quad slice^{sm_{v_j}}_{sc} := wellformSlice(sm_{v_j}, slice^{sm_{v_j}}_{sc})$;
16 **return** $slice^{sm_{v_j}}_{sc}, \Delta^{slice_{sc}}_{v_i,v_j}$;

Algorithm 4.11.: Function `initSliceRegDelta`

Input: Previous State Machine Slice $slice^{sm_{v_i}}_{sc}$ and Dependency Graph Regression Delta $\Delta^{DG}_{v_i,v_j}$

Output: Initialized Slice Regression Delta $\Delta^{slice_{sc}}_{v_i,v_j}$

1 **Function** *initSliceRegDelta*
2 $\quad$ **forall** $op^{DG} \in \Delta^{DG}_{v_i,v_j}$ **do**
3 $\quad\quad$ **if** $op^{DG} ==$ rem n **then**
4 $\quad\quad\quad elem_n := getElement(slice^{sm_{v_i}}_{sc}, n)$;
5 $\quad\quad\quad$ **if** $elem_n \neq \dashv$ **then**
6 $\quad\quad\quad\quad \Delta^{slice_{sc}}_{v_i,v_j} := \Delta^{slice_{sc}}_{v_i,v_j} \cup \{\mathsf{rem}\ elem_n\}$;
7 **return** $\Delta^{slice_{sc}}_{v_i,v_j}$;

computation in a subsequent iteration. In contrast, if the element *elem'* is not contained in the slice, we (1) derive a respective add operation and record the change operation in the set $\Delta^{elem}_{slice_{sc}}$ and (2) extend the set $Elem_{Next}$ by *elem'*. The set $\Delta^{elem}_{slice_{sc}}$ captures all slice change operations which are determined for the state machine element *elem* in the current iteration of the slice computation.

Afterwards, we determine those elements *elem'* which have to be added to the slice due to the reachability check (cf. Line 9) similarly as determined by the function `checkReachability` of the

Algorithm 4.12.: Function checkDependency

Input: State Machine Element *elem*, State Machine sm_{v_j}, Intermediate State Machine Slice $slice_{sc}^{sm_{v_j}}$, Dependency Graph $DG_{sm_{v_j}}$, and Element Set $Elem_{Next}$

Output: Set of Slice Change Operation $\Delta_{slice_{sc}}^{elem}$

```
1  Function checkDependency
2      n_elem := getNode(N_DG_sm_vj, elem);
3      forall dep ∈ Dep_DG_sm_vj do
4          if dep == (n_elem, l_dep, n_elem') then
5              elem' := getElement(sm_vj, n_elem');
6              r := getRegion(sm_vj, slice_sc^sm_vj, elem');
7              if typeOf(elem') == STATE ∧ !isInitState(r, elem') then
8                  if elem' ∉ S_r then
9                      Δ_slice_sc^elem := Δ_slice_sc^elem ∪ {add elem'};
10                 Elem_Next := Elem_Next ∪ {elem'};
11             else if typeOf(elem') == STATE ∧ isInitState(r, elem') then
12                 Elem_Next := Elem_Next ∪ {elem'};
13             else if typeOf(elem') == TRANSITION then
14                 if elem' ∉ T_r then
15                     Δ_slice_sc^elem := Δ_slice_sc^elem ∪ {add elem'};
16                 Elem_Next := Elem_Next ∪ {elem'};
17     return Δ_slice_sc^elem;
```

standard slicing computation shown in Alg. 4.4. As depicted in Alg. 4.13, we again examine if those elements *elem'* are already contained in the slice or not and, therefore, extend the set $Elem_{Next}$ by *elem'* and potentially the set $\Delta_{slice_{sc}}^{elem}$ with add operations, correspondingly. Just as for the standard slice computation which is described above (cf. Sect. 4.1.1), we extend the set $Elem_{Next}$ solely if an element was not already comprised in preceding iterations or skip the extension, otherwise. As last step of each iteration, we apply the set $\Delta_{slice_{sc}}^{elem}$ to adapt the (intermediate) slice $slice_{sc}^{sm_{v_j}}$ and further integrate the new determined slice change operations from $\Delta_{slice_{sc}}^{elem}$ into the slice regression delta $\Delta_{v_i,v_j}^{slice_{sc}}$. In case the set $Elem_{Next}$ is not empty, we start a new iteration of the incremental slice computation.

After finishing the iterations, the slice $slice_{sc}^{sm_{v_j}}$ may contain invalid and, thus, obsolete state machine elements from the previous slice $slice_{sc}^{sm_{v_i}}$ to be also removed. Those elements are not integrated into the recomputed slice during the incremental computation, e.g., due to the removal of a respective dependency in the dependency graph adaptation. Hence, we derive slice change operations to remove obsolete state machine elements $elems' \in (Elems_{slice_{sc}^{sm_{v_i}}} \setminus Elems_{slice_{sc}^{sm_{v_j}}})$ (cf. Line 12), where $Elems_{slice_{sc}^{sm_{v_i}}}$ denotes the set of state machine elements contained in slice $slice_{sc}^{sm_{v_i}}$ and $Elems_{slice_{sc}^{sm_{v_j}}}$ the set of elements of the recomputed slice $slice_{sc}^{sm_{v_j}}$ solely comprising those elements which are integrated during the computation. The captured remove operations $\Delta_{slice_{sc}}^{obsolete}$ are then applied to the slice $slice_{sc}^{sm_{v_j}}$ and further the slice regression delta $\Delta_{v_i,v_j}^{slice_{sc}}$ is extended by those operations. As

Algorithm 4.13.: Function `checkReachability`

Input: State Machine Element *elem*, State Machine sm_{v_j}, Intermediate State Machine Slice $slice_{sc}^{sm_{v_j}}$, Element Set $Elem_{Next}$, and Set of Slice Change Operations $\Delta_{slice_{sc}}^{elem}$

Output: Updated Set of Slice Change Operations $\Delta_{slice_{sc}}^{elem}$

1 **Function** *checkReachability*
2 $r_{sm_{v_j}} := getRegion(sm_{v_j}, elem)$;
3 $r_{slice_{sc}^{sm_{v_j}}} := getRegion(sm_{v_j}, slice_{sc}^{sm_{v_j}}, elem)$;
4 **if** $typeOf(elem) == STATE$ **then**
5 **forall** $t \in T_{r_{sm_{v_j}}}$ **do**
6 **if** $hasTargetState(t, elem)$ **then**
7 **if** $t \notin T_{r_{slice_{sc}^{sm_{v_j}}}}$ **then**
8 $\Delta_{slice_{sc}}^{elem} := \Delta_{slice_{sc}}^{elem} \cup \{\mathsf{add}\, t\}$;
9 $Elem_{Next} := Elem_{Next} \cup \{t\}$;
10 **else**
11 $s_{source} := getSourceState(elem)$;
12 $s_{target} := getTargetState(elem)$;
13 **if** $s_{target} \notin S_{r_{slice_{sc}^{sm_{v_j}}}}$ **then**
14 $\Delta_{slice_{sc}}^{elem} := \Delta_{slice_{sc}}^{elem} \cup \{\mathsf{add}\, s_{target}\}$;
15 **if** $s_{source} \notin S_{r_{slice_{sc}^{sm_{v_j}}}}$ **then**
16 $\Delta_{slice_{sc}}^{elem} := \Delta_{slice_{sc}}^{elem} \cup \{\mathsf{add}\, s_{source}\}$;
17 **else**
18 $Elem_{Next} := Elem_{Next} \cup \{s_{source}\}$;
19 **return** $\Delta_{slice_{sc}}^{elem}$;

last step of the incremental slice computation, we ensure well-formedness by applying the function `wellformSlice` as depicted in Alg. 4.5.

In the end, the algorithm of the incremental slice computation shown in Alg. 4.10 terminates as we reach a fix-point, i.e., no further element can be added to the slice, which is represented by the emptiness of the set $Elem_{Next}$. Just as for the standard slicing process (cf. Sect. 4.1.1), we focus on states as well as transitions and abstract from regions during the recomputation, but obtain as result a reduced, yet well-formed state machine (cf. Def. 3.8).

We summarize the list of symbols used for the definition of our incremental state machine slicing in Tab. 4.2. To recapitulate, we incrementally adapt a dependency graph $DG_{sm_{v_i}}$ of a previously analyzed variant v_i to obtain the dependency graph $DG_{sm_{v_j}}$ of the current variant v_j under analysis by applying dependency graph change operations $op^{DG} \in \Delta_{v_i,v_j}^{DG}$ captured in a dependency graph regression delta. The incremental application of the change operations is defined by the dependency graph delta application function apply_{DG}. We use the dependency graph regression delta as starting point for the incremental slice computation. During the recomputation of a slice, we capture slice change operations $op^{Slice} \in \Delta_{v_i,v_j}^{slice_{sc}}$ in a slice regression delta representing the differences between a

Table 4.2: Symbol Summary of Incremental State Machine Slicing Definition

Symbol	Description
$op^{DG}, \mathcal{OP}^{DG}$	Dependency graph change operation; Universe of dependency graph change operation
$\Delta^{DG}_{v_i,v_j}$	Dependency graph regression delta
apply_{DG}	Dependency graph delta application function
$op^{Slice}, \mathcal{OP}^{Slice}$	Slice change operation; Universe of slice change operations
$\Delta^{slice_{sc}}_{v_i,v_j}$	Slice regression delta
apply_{slice}	Slice delta application function

previous slice w.r.t. a slicing criterion and the recomputed slice. The application of the slice change operations is defined by the slice delta application function apply_{slice}. We exploit the slice regression delta to reason about change impact between two consecutively tested variants and version of variants as discussed in the next paragraph.

Example 4.3: Incremental Slice Computation

Consider the sample state machine slice $slice^{sm_{v_{core}}}_{t_2}$ of the core state machine $sm_{v_{core}}$ for transition t_2 from Ex. 4.1 depicted in Fig. 4.1b again. When stepping from the core variant v_{core} to variant v_1, we recompute the slice for the transition. We start the recomputation by exploiting the determined dependency graph regression delta $\Delta^{DG}_{v_{core},v_1}$ from Ex. 4.2. We first examine the regression delta $\Delta^{DG}_{v_{core},v_1}$ for remove operations of element nodes. The regression delta comprises solely one remove operation of an element node, i.e., $\mathsf{rem}\, t_8$, which is not incorporated in the slice computation as the transition t_8 was not part of the core slice $slice^{sm_{v_{core}}}_{t_2}$. Afterwards, we initialize the set $Elem_{Next}$ with the slicing criterion transition t_2 and start the iterations. In the first iteration, we determine the state machine elements to be added to the slice based on dependencies to t_2 captured in the dependency graph $DG_{sm_{v_1}}$. Based on the existing dependency (t_2, tcd, t_1) as well as the new dependencies (t_2, tcd, t_{10}) and (t_2, tcd, t_{12}), we integrate the three transitions t_1, t_{10}, and t_{12} into the slice and extend the set $Elem_{Next}$. In contrast to transition t_1 already comprised in slice $slice^{sm_{v_{core}}}_{t_2}$, we derive add operations for transitions t_{10} and t_{12} and extend the set $\Delta^{t_2}_{slice_{t_2}}$. As second step, we check the reachability of t_2 and integrate the states a_1 and a_2. In the remaining iterations over the set $Elem_{Next}$, the states d_1, d_2, d_3, and s_0 as well as the transitions t_9 and t_{11} are added to the slice via respective add change operations. The resulting slice $slice^{sm_{v_1}}_{t_2}$ is shown in Fig. 4.4, where the differences to the core slice $slice^{sm_{v_{core}}}_{t_2}$, i.e., the additions of state machine elements, are marked with a $+$. We capture those differences in the slice regression delta $\Delta^{slice_{t_2}}_{v_{core},v_1}$.

Application for Change Impact Analysis.

In general, our incremental slicing technique facilitates incremental analysis in SPLE [PBvdL05] to support other development activities such as maintenance [Wei81; ACH+13] in order to tackle the

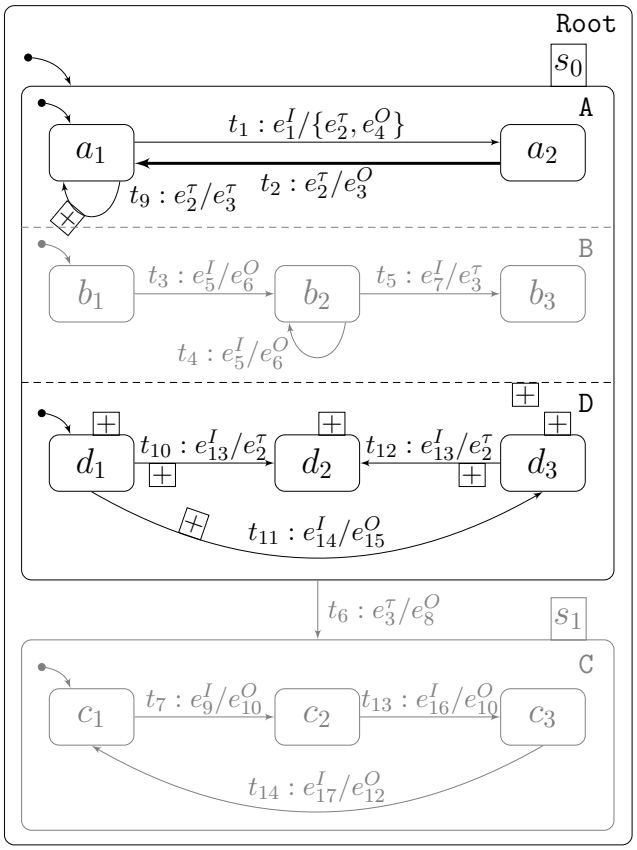


Figure 4.4: Recomputed State Machine Slice $slice_{t_2}^{sm_{v_1}}$ for Transition t_2 Including Slice Differences to Core State Machine Slice $slice_{t_2}^{sm_{v_{core}}}$

challenge of increasing complexity of nowadays variant-rich software systems. In this thesis, we apply our technique for change impact analysis to support our model-based SPL regression testing framework by guiding the retest test selection (cf. Chapt. 5).

By stepping from one variant v_i under test to the subsequent variant v_j during incremental testing of a certain SPL version which is described in Sect. 5.1, we examine the differences between both variants captured as state machine regression delta Δ_{v_i,v_j} to identify their impact to already tested behavior by means of changed (inter)dependencies of software parts. As already mentioned, we focus on backward slicing such that a slice solely comprises those state machine elements that have a potential influence on a slicing criterion during execution. Hence, our technique determines changes of the dependencies, i.e., slice differences, during the incremental slice computation that may influence the already tested behavior of the state machine element denoting the slicing criterion. The slice differences indicate retest potentials, i.e., already tested behavior to be revalidated during regression testing [GB08; YH12]. In case, the previous slice $slice_{sc}^{sm_{v_i}}$ of v_i as well as the recomputed slice $slice_{sc}^{sm_{v_j}}$ for the current variant v_j under test are identical, we obtain an empty slice regression delta $\Delta_{v_i,v_j}^{slice_{sc}}$. That means, both variants share the same unchanged behavior w.r.t. the slicing criterion and no retest of test cases is required [Bin98].

For an automated reasoning about retest decisions, we require a scale by means of an expressive criterion incorporating slice differences. In the context of software testing, test adequacy criteria are

used to guide several test activities [AO16; UL06], e.g., the test-case generation. Therefore, adequacy criteria are also promising scales to guide our retest test selection. We adopt the concept of adequacy criteria by defining a retest test coverage criterion which is described in Sect. 5.2.1.

Furthermore, our incremental state machine slicing is also applicable for change impact analysis between versions of variants of consecutive SPL versions under test. We, therefore, capture the differences between both versions of a variant also as a state machine regression delta $\Delta_{v_i^{\theta_k}, v_i^{\theta_{k+1}}}$, where the delta sets of the old $\Delta_{v_i^{\theta_k}}$ and of the modified version $\Delta_{v_i^{\theta_{k+1}}}$ are incorporated for the delta derivation. The regression delta $\Delta_{v_i^{\theta_k}, v_i^{\theta_{k+1}}}$ is then used as a basis to apply our incremental slicing technique exploiting the commonality between both versions and to detect changed behavior to be retested by our regression testing framework after evolution occurs (cf. Chapt. 5). However, the identification of modified or even unchanged variants is another challenge to be explored by a change impact analysis technique. We tackle this challenge by reasoning about the higher-order delta application to investigate changes to the variant set in terms of new, modified, and unchanged variants when stepping to the next SPL version to be tested. We describe the reasoning and, therefore, the change impact analysis of versions of variants in the next section.

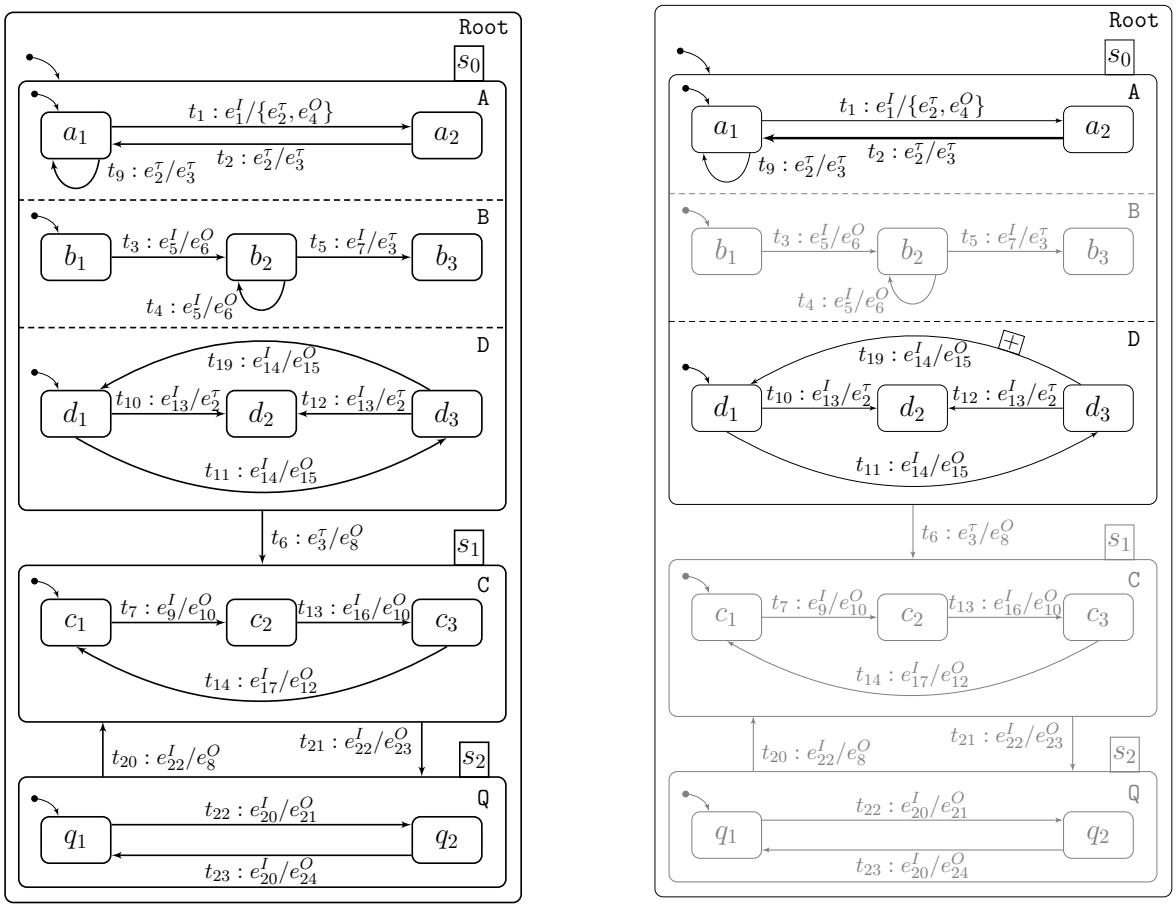
Example 4.4: Slicing-Based Change Impact Analysis

Consider the recomputed state machine slice $slice_{t_2}^{sm_{v_1}^{\theta_0}}$ of state machine $sm_{v_1}^{\theta_0}$ for transition t_2 from Ex. 4.3 depicted in Fig. 4.4 again. The slice differences captured in the slice regression delta $\Delta_{v_{core}, v_1}^{slice_{t_2}}$ indicate the impact of model changes applied when stepping from the core variant v_{core} to the variant v_1 to be tested.

In addition, after finishing the test of SPL version θ_0, we step to the next SPL version θ_1 to be tested (cf. Chapt. 5). Assume that we identify the modification of variant $v_1^{\theta_0}$ and its state machine $sm_{v_1}^{\theta_0}$ to variant $v_1^{\theta_1}$ with state machine $sm_{v_1}^{\theta_1}$ which is shown in Fig. 4.5a. By taking the respective delta sets $\Delta_{v_1^{\theta_0}} = \{\delta_1, \delta_2, \delta_3\}$ and $\Delta_{v_1^{\theta_1}} = \{\delta_1', \delta_2, \delta_3, \delta_6\}$ into account, we derive the state machine regression delta $\Delta_{v_1^{\theta_0}, v_1^{\theta_1}} = \{\mathsf{add}\, t_{19}, \mathsf{add}\, t_{20}, \mathsf{add}\, t_{21}, \mathsf{add}\, s_2, \mathsf{add}\, \mathsf{Q}, \mathsf{add}\, (s_2, \mathsf{Q})\}$. The regression delta $\Delta_{v_1^{\theta_0}, v_1^{\theta_1}}$ is then used to apply our slicing technique, where the slice for transition t_2 is recomputed. The result is depicted in Fig. 4.5b, where the difference between both slices, i.e., the addition of transition t_{19} is marked with a $+$. We capture this difference in a respective slice regression delta $\Delta_{v_1^{\theta_0}, v_1^{\theta_1}}^{slice_{t_2}}$ and exploit it to guide our retest test selection for testing the modified variant version $v_1^{\theta_1}$.

4.2 Change Impact Analysis of Versions of Variants

SPL evolution emerges, e.g., from changing requirements and affects the development artifacts and their variant-specific composition due to necessary changes [SB99; MSC14; BP14]. As a consequence, the variant set $\mathbb{V}_{\theta_i} = \{v_1, \ldots, v_n\}$ of SPL version θ_i also changes to $\mathbb{V}_{\theta_{i+1}} = \{v_1, v_3, v_4', \ldots, v_l\}$ of version θ_{i+1} in terms of added, removed, modified, and unchanged variants. From a regression testing point of view [YH12], modified and unchanged variants are of special interest as (1) modifications indicate retest potentials between two versions of an already tested variant, and (2) unchanged variants does not require to be retested as no retest potentials arise. Therefore, to apply retest test



(a) State Machine $sm_{v_1}^{\theta_1}$

(b) State Machine Slice $slice_{t_2}^{sm_{v_1}^{\theta_1}}$

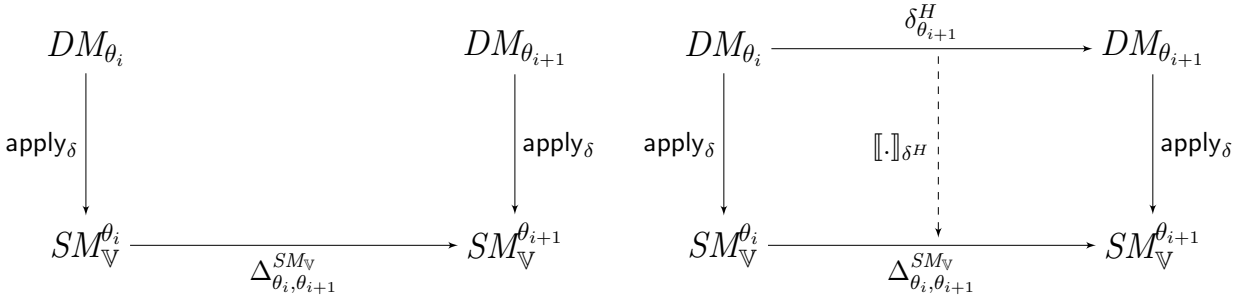
Figure 4.5: (a) State Machine $sm_{v_1}^{\theta_1}$ of Variant $v_1^{\theta_1}$ in SPL Version θ_1; (b) Recomputed State Machine Slice $slice_{t_2}^{sm_{v_1}^{\theta_1}}$ of State Machine $sm_{v_1}^{\theta_1}$ for Transition t_2 in SPL Version θ_1 Including Slice Differences to State Machine Slice $slice_{t_2}^{sm_{v_1}^{\theta_0}}$ of State Machine $sm_{v_1}^{\theta_0}$ in SPL Version θ_0

selection after SPL evolution (cf. Chapt. 5), we are interested in how the variant set changes when stepping to the next SPL version under test.

To determine the performed variant set changes, we can follow a naive approach, where we compare the variant set $\mathbb{V}_{\theta_i}$ of the prior SPL version θ_i with its evolved version $\mathbb{V}_{\theta_{i+1}}$ in a product-by-product way leading to an inefficient impact analysis [TAK+14]. Furthermore, the product-by-product comparison is challenging as changes performed in the problem as well as solution space may prevent from (1) an unambiguous mapping between a variant and its modified version or (2) an automated classification by means of added, removed, and modified variants. For instance, by renaming features to alter the feature model fm_{θ_i} of SPL version θ_i to obtain $fm_{\theta_{i+1}}$, the set of derivable feature configurations $F_{\mathbb{V}}^{\theta_i}$ changes to $F_{\mathbb{V}}^{\theta_{i+1}}$, where the renaming can either be identified as a modification of feature configurations or can be interpreted as the removal and addition of non-related feature configurations. The same holds for the solution space. By comparing the set of variant-specific state machines $SM_{\mathbb{V}}^{\theta_i}$ of SPL version θ_i and $SM_{\mathbb{V}}^{\theta_{i+1}}$ of SPL version θ_{i+1}, it is not directly inferable which state machine $sm \in SM_{\mathbb{V}}^{\theta_{i+1}}$ represents the modification of a state machine

$sm \in SM_{\mathbb{V}}^{\theta_i}$ in the previous SPL version θ_i. Hence, the incorporation of changes applied in an evolution step is crucial to reason about the change impact on the variant set of an SPL version.

Existing techniques providing such kind of impact analysis [NSS16; SBT16; TBK09; BKL+16] either detect changes to the variant set on the feature model level [NSS16; TBK09; BKL+16] or identify that a variant is modified [SBT16], but do not determine how the original version and the modified version of the variant differ. We cannot exploit those techniques for our model-based regression testing framework (cf. Chapt. 5) as (1) we perform retest test selection based on identified retest potentials on the state machine test model level and, hence, require the change categorization to be made in the solution space, and (2) we require the information how modified versions of variants differ to facilitate their efficient retesting. In this thesis, we exploit the application of higher-order delta modeling (cf. Sect. 3.3) such that we are able to derive and reason about changes to the variant set between consecutively tested SPL versions. Higher-order deltas $\delta_{\theta}^{H} \in DM_{\Theta}^{H}$ specify how the delta set Δ_{DM_θ} of a version-specific delta model DM_θ changes in terms of additions, removals, and modifications of state machine deltas. By taking those changes into account, we infer respective changes on variant-specific delta sets Δ_v and, therefore, (1) directly provide the difference between modified versions of variants and (2) facilitate the categorization by means of added, removed, modified, and unchanged variants. Based on this reasoning process, we are able to abstract from feature model evolution as a categorization of respective feature configurations does not provide further information regarding the change of the variant set. In addition, we do not have to generate and compare variant-specific state machine test models to obtain the change categorization.



(a) Naive Change Impact Analysis

(b) Delta-Oriented Change Impact Analysis

Figure 4.6: Overview of the Reasoning about Higher-Order Delta Application

4.2.1 Delta-Oriented Evolution of Variant Sets

Following the naive approach as shown in Fig. 4.6a, we would compare the sets of variant-specific state machines $SM_{\mathbb{V}}^{\theta_i}$ and $SM_{\mathbb{V}}^{\theta_{i+1}}$ to derive how the variant has changed which can be captured as variant set evolution delta $\Delta_{\theta_i,\theta_{i+1}}^{SM_{\mathbb{V}}}$. In contrast, we are able to derive $\Delta_{\theta_i,\theta_{i+1}}^{SM_{\mathbb{V}}}$ by analyzing the application of the higher-order delta $\delta_{\theta_{i+1}}^{H}$ via the reasoning function $[\![.]\!]_{\delta^H}$ as depicted in Fig. 4.6b. The reasoning function $[\![.]\!]_{\delta^H}$ is defined such that we directly pass on the changes captured in the higher-order delta $\delta_{\theta_{i+1}}^{H}$ to the variant-specific delta sets $\Delta_v \in \Delta_{\mathbb{V}_{\theta_i}}$ of the previous SPL version to obtain the altered and categorized set $\Delta_{\mathbb{V}_{\theta_{i+1}}}$ when stepping to the next SPL version to be tested. By $\Delta_{\mathbb{V}_\theta}$, we refer to the set of all variant-specific delta sets of an SPL version θ. Based on the mapping between variant-specific state machines sm_v and their corresponding variant-specific delta sets Δ_v which

are potentially altered by the higher-order delta application, we are able to derive the variant set evolution delta $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$ capturing the addition, removal, or modification of the variant-specific state machines sm_v. In the following, we provide the definition of the variant set evolution delta $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$ used as input in our model-based regression testing framework to guide the retest of versions of variants when stepping to the next SPL version to be tested (cf. Chapt. 5). Afterwards, in the next section, we introduce the (incremental) delta set derivation process that allows for the reasoning about the higher-order delta application and, therefore, define the reasoning function $[\![.]\!]_{\delta^H}$.

To capture the changes of the variant set facilitating a categorization by means of added, removed, modified, and unchanged variants, we, again, adopt the concept of delta modeling [Sch10; CHS15]. Accordingly, we define *variant set change operations* on the state machine level such that a variant-specific state machine sm_v can be added to, modified, or removed from the set $SM^{\theta}_{\mathbb{V}}$ of all state machine variants of an SPL version θ derivable from the version-specific delta model DM_{θ}.

Definition 4.7: Variant Set Change Operation

Let $\mathcal{OP}^{\mathcal{SM}}$ be the universe of all variant set change operations which, in turn, is defined over the universe $\mathcal{SM}$ of all well-formed state machines. A *variant set change operation* $op^{\mathcal{SM}} \in \mathcal{OP}^{\mathcal{SM}}$ defines one of the following transformations:

- add sm, i.e., a state machine $sm \in \mathcal{SM}$ is added,
- rem sm, i.e., a state machine $sm \in \mathcal{SM}$ is removed, and
- mod (sm, sm'), i.e., a state machine $sm \in \mathcal{SM}$ is modified to state machine $sm' \in \mathcal{SM}$.

When stepping to the next SPL version, we encapsulate the performed variant set change operations in a respective *variant set evolution delta*. By applying a variant set evolution delta to the state machine set $SM^{\theta_i}_{\mathbb{V}}$ of the prior tested SPL version θ_i, we obtain the valid, yet updated state machine set $SM^{\theta_{i+1}}_{\mathbb{V}}$ of the next SPL version θ_{i+1} under test.

Definition 4.8: Variant Set Evolution Delta

Let $\mathsf{apply}_{\mathcal{SM}}$ be the incremental application function to transform a version-specific state machine set $SM^{\theta_i}_{\mathbb{V}}$ into the subsequent version-specific state machine set $SM^{\theta_{i+1}}_{\mathbb{V}}$ based on a given set of variant set change operations. A *variant set evolution delta* $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}} = \{op^{\mathcal{SM}}_1, \ldots, op^{\mathcal{SM}}_m\}$ captures all variant set change operations such that $SM^{\theta_{i+1}}_{\mathbb{V}} = \mathsf{apply}_{\mathcal{SM}}(SM^{\theta_i}_{\mathbb{V}}, \Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}})$ holds.

The incremental application function $\mathsf{apply}_{\mathcal{SM}}$ is defined similar to the other incremental delta application functions already introduced in Chapt. 3. For each evolution step from SPL version θ_i to version θ_{i+1}, a respective variant set evolution delta $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$ exist. We refer to Lity et al. [LKS16] for the proof of the existence.

In Tab. 4.3, we summarize the list of symbols used for the definition of the delta-oriented evolution of variant sets. To recapitulate, we reason about the higher-order delta application via the reasoning function $[\![.]\!]_{\delta^H}$ to determine changes of the variant set $SM^{\theta_i}_{\mathbb{V}}$ of the previous SPL version θ_i represented by the set of variant-specific state machines $sm_v \in SM^{\theta_i}_{\mathbb{V}}$ to obtain the variant set $SM^{\theta_{i+1}}_{\mathbb{V}}$ of the subsequent SPL version θ_{i+1}. Those variant set change operations $op^{\mathcal{SM}}$ specify the addition, the removal, or the modification of variant-specific state machines and are captured in a variant set evolution delta $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$. Hence, we obtain $SM^{\theta_{i+1}}_{\mathbb{V}} = \mathsf{apply}_{\mathcal{SM}}(SM^{\theta_i}_{\mathbb{V}}, \Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}})$ by applying the variant

Table 4.3: Symbol Summary of Delta-Oriented Variant Set Evolution

Symbol	Description
$op^{\mathcal{SM}}, \mathcal{OP}^{\mathcal{SM}}$	Variant set change operation; Universe of variant set change operations
$\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$	Variant set evolution delta
$\mathsf{apply}_{\mathcal{SM}}$	Variant set evolution delta application function
$[\![.]\!]_{\delta^H}$	Higher-oder delta application reasoning function

set evolution delta $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$ to the previous variant set $SM^{\theta_i}_{\mathbb{V}}$ via the application function $\mathsf{apply}_{\mathcal{SM}}$. In order to derive the variant set change operations between two SPL versions and, therefore, to reason about the impact of the higher-order delta application, we exploit the evolution change operations captured in the higher-order delta $\delta^H_{\theta_{i+1}}$ and directly pass on those changes to the variant-specific delta sets Δ_v based on their incremental adaptation as described in the next section.

4.2.2 Delta Set Derivation

As we perform the reasoning on the set $\Delta_{\mathbb{V}_\theta}$ of all variant-specific delta sets Δ^θ_v of SPL version θ, we require $\Delta_{\mathbb{V}_\theta}$ to be known in advance. To determine all delta sets, we can iterate over all feature configurations $F_v \in F^\theta_{\mathbb{V}}$ and evaluate which state machine deltas $\delta \in \Delta^\theta_{DM}$ have to be applied for the respective variant v based on their application conditions φ_δ. However, if evolution occurs solely in the problem space, e.g., based on the refactoring of the feature model [TBK09; BKL+16], we have to regenerate all delta sets based on the updated set of feature configurations. In such a case, the categorization of variants requires more effort and also gets rather difficult, e.g., we potentially miss the identification of modifications of delta sets between the subsequent SPL versions resulting in false categorizations in terms of removals and additions of the respective variants. Therefore, we propose a delta set derivation that is independent from the feature configurations $F^\theta_{\mathbb{V}}$ of an SPL version and that facilitates the incremental adaptation and categorization of delta sets based on the application of a higher-order delta $\delta^H_{\theta_{i+1}}$ when stepping to the next SPL version under test.

In Fig. 4.7, we provide an overview of our (incremental) delta set derivation which is slightly similar to the general process of incremental model slicing defined in Sect. 4.1.2 and shown in Fig. 4.2. As depicted on the left hand side, we start from the delta model of the current SPL version θ_i and perform a *delta dependency analysis*, where we examine how state machine deltas $\delta \in \Delta_{DM_\theta}$ are related to each other regarding their potential combinations in variant-specific delta sets. The result of the dependency analysis in terms of delta dependencies is captured in a *delta dependency graph*.

Afterwards, we use the delta dependency graph to generate a *variant tree* capturing all derivable delta sets in a compact way as variant tree paths. A variant tree also comprises for each non-derivable delta set its set of restrictions defined by unfulfilled delta dependencies. Based on this representation, we ensure the traceability how and which delta sets may change due to the application of a higher-order delta. When stepping to the next SPL version to be tested, we apply a similar process as for our incremental slicing technique described in Sect. 4.1.2. First, we incrementally adapt the delta dependency graph by incorporating the evolution changes captured in a higher-order delta (cf. Fig. 4.7 (1)) and derive a delta dependency graph regression delta. Second, we exploit

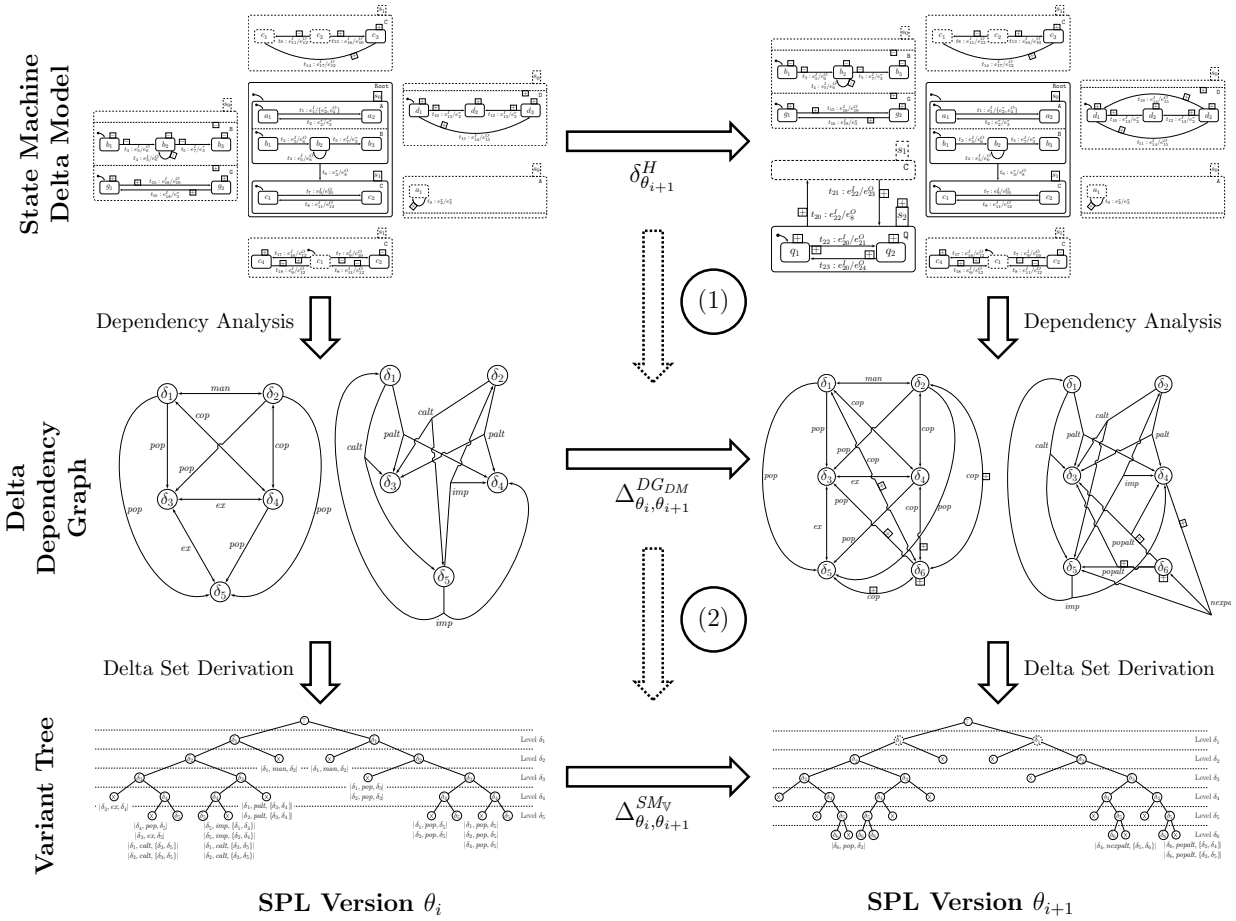


Figure 4.7: Overview of the Incremental Delta Set Derivation

the delta dependency graph regression delta to recompute the variant tree for the next SPL version (cf. Fig. 4.7 (2)). During the recomputation, i.e., the update of the variant-specific delta sets, we categorize if variants are added, modified, or unchanged and, thus, reason about the change impact of the higher-order delta application and further derive the variant set evolution delta.

Delta Dependency Analysis. By analyzing how state machine deltas $\delta \in \Delta_{DM_\theta}$ of the delta model DM_θ of an SPL version θ are related to each other w.r.t. their potential combination in variant-specific delta sets, we determine their dependencies restricting or predefining their combinations and capture them in a delta dependency graph. A delta dependency graph is similar defined to a state machine dependency graph (cf. Def. 4.2) and, hence, contains (1) a set of *delta nodes* such that for each delta $\delta \in \Delta_{DM_\theta}$ of the delta model DM_θ a respective delta node exists, and (2) a set of *dependency edges* connecting delta nodes representing that the respective deltas are related w.r.t. their combination, e.g., two deltas are always encapsulated together in variant-specific delta sets.

Definition 4.9: Delta Dependency Graph

Let $\mathcal{L}_{\Delta_{Dep}}$ be the set of delta dependency labels represented by the names of the dependencies taken into account for the delta dependency analysis. A *delta dependency graph* $DG_{DM} =$

(N_{DM}, Dep_{DM}) is a hypergraph defined over the label set $\mathcal{L}_{\Delta_{Dep}}$, where
- $N_{DM} = \{n_1, \ldots, n_n\}$ is a finite set of delta nodes, and
- $Dep_{DM} \subseteq N_{DM} \times \mathcal{L}_{\Delta_{Dep}} \times N_{DM}^{+}$ is a delta dependency edge relation.

To determine whether state machine deltas $\delta \in \Delta_{DM}$ are related to each other, we investigate their potential combination as well as independent incorporation in a variant-specific delta set Δ_v of a variant $v \in \mathbb{V}$. Therefore, we use their application conditions φ_δ and apply satisfiability checks w.r.t. the feature model *fm* of the SPL under consideration. Please note that in case we test solely a representative subset $\mathbb{V}'_\theta \subseteq \mathbb{V}_\theta$ of variants of an SPL version θ determined by applying sampling strategies [VAT+18], we perform the satisfiability checks w.r.t. the disjunction of the respective feature configurations $F^\theta_{\mathbb{V}'}$. The satisfiability checks are sufficient for the examination of delta dependencies such that we do not need the variant-specific feature configurations $F_v \in F_{\mathbb{V}}$. As already mentioned, the abstraction from feature configurations is a benefit of our analysis as we (1) do not require all valid feature configurations in advance, and (2) also are independent from potential changes to feature configurations which do not have an affect on variant-specific delta sets. Obviously, the number of performed satisfiability checks increases with the number of deltas to be analyzed and their potential combinations. In general, solving a satisfiability problem is NP-complete. However, for the analysis of feature models to which our delta dependency analysis belongs to, the solving of a respective satisfiability problem scales very well [MWC09; LGC+15; BSR10] and, therefore, our analysis is not impeded by the application of satisfiability checks.

For the definition of the delta dependency edge relation Dep_{DM}, we introduce distinct delta dependencies $\mathcal{L}_{\Delta_{Dep}} = \mathcal{L}^{single}_{\Delta_{Dep}} \cup \mathcal{L}^{multi}_{\Delta_{Dep}}$ classified as single- or multi-target dependencies. A *single-target dependency* denotes a one-to-one relation between two deltas $\delta, \delta' \in \Delta_{DM}$, i.e., it captures the potential combination of two deltas, and is represented by a dependency edge connecting the respective delta nodes $n_\delta, n_{\delta'} \in N_{DM}$. In contrast, a *multi-target dependency* denotes a one-to-many relation between a source delta $\delta \in \Delta_{DM}$ and some target deltas $\delta', \delta'' \in \Delta_{DM}$, i.e., it captures the potential combination of at least three deltas, and is represented as hyper edge connecting their delta nodes $n_\delta, n_{\delta'}, n_{\delta''} \in N_{DM}$. For the derivation of single-target dependencies, we solely take the application conditions of the two deltas under analysis into account and check whether the combination is satisfiable w.r.t. the feature model *fm*, i.e., there exists at least one variant-specific feature configuration that satisfies the feature model as well as the application conditions. For the derivation of multi-target dependencies, we mainly incorporate the determined single-target dependencies $(n_\delta, l^{single}_{\Delta_{Dep}}, n_{\delta'}), (n_\delta, l^{single}_{\Delta_{Dep}}, n_{\delta''}), (n'_\delta, l^{single}_{\Delta_{Dep}}, n_{\delta''}) \in Dep_{DM}$ with $l^{single}_{\Delta_{Dep}} \in \mathcal{L}^{single}_{\Delta_{Dep}}$. In some cases, we further have to take the application conditions w.r.t. the feature model *fm* into account to exclude invalid delta combinations. Based on the systematic examination of possible combinations of two state machine deltas, we identified four single-target dependencies $\mathcal{L}^{single}_{\Delta_{Dep}} = \{man, ex, pop, cop\}$ which are defined as follows, where $[\![fm]\!]_{\mathbb{B}}$ refers to the representation of a feature model *fm* as propositional formula [Man02; Bat05; BSR10] and $[\![F]\!]$ denotes the feature selection function for feature configurations defined in Sect. 3.2:

- **Mandatory Dependency** (*man*) – Two deltas $\delta, \delta' \in \Delta_{DM}$ are *mandatory dependent* represented as $(n_\delta, man, n_{\delta'})$, iff there is no variant $v \in \mathbb{V}$ such that δ is applied without δ' and vice versa in a respective delta set Δ_v, i.e., both deltas are always applied together:

$$(n_\delta, man, n_{\delta'}) \in Dep_{DM} \Leftrightarrow (\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \varphi_{\delta'}) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \neg\varphi_\delta \wedge \varphi_{\delta'}) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'})$$

- **Exclusive Dependency** (*ex*) – Two deltas $\delta, \delta' \in \Delta_{DM}$ are *exclusive dependent* denoted as $(n_\delta, ex, n_{\delta'})$, iff there is no variant $v \in \mathbb{V}$ such that δ and δ' are applied together in a delta set Δ_v:

$$(n_\delta, ex, n_{\delta'}) \in Dep_{DM} \Leftrightarrow (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \varphi_{\delta'})$$

- **Partial Optional Dependency** (*pop*) – Two deltas $\delta, \delta' \in \Delta_{DM}$ are *partial optional dependent* represented as $(n_\delta, pop, n_{\delta'})$, iff δ is independently applicable for a variant $v \in \mathbb{V}$ w.r.t. δ', but δ' is not applicable without δ in a delta set Δ_v, i.e., delta δ' requires the application of delta δ:

$$(n_\delta, pop, n_{\delta'}) \in Dep_{DM} \Leftrightarrow (\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \varphi_{\delta'}) \wedge (\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'}) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \neg\varphi_\delta \wedge \varphi_{\delta'})$$

- **Complete Optional Dependency** (*cop*) – Two deltas $\delta, \delta' \in \Delta_{DM}$ are *complete optional dependent* denoted as $(n_\delta, cop, n_{\delta'})$, iff both deltas δ and δ' are independently applicable for variants $v \in \mathbb{V}$ in their respective delta sets Δ_v:

$$(n_\delta, cop, n_{\delta'}) \in Dep_{DM} \Leftrightarrow (\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \varphi_{\delta'}) \wedge (\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'}) \wedge (\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \neg\varphi_\delta \wedge \varphi_{\delta'})$$

In addition, we identified eight multi-target dependencies $\mathcal{L}^{multi}_{\Delta_{Dep}} = \{calt, palt, copalt, popalt, opalt, nexalt, nexpalt, imp\}$ also deduced by systematically examining the combinations of at least three deltas. However, due to the exponential number of possible combinations, we further investigated the three delta-oriented subject SPLs and their versions (cf. Sect. 3.4) to determine what combination alternatives exist. Based on this investigation, we were able to reduce the number of combinations to those eight alternative dependencies. Please note, we do not consider this catalog $\mathcal{L}^{multi}_{\Delta_{Dep}}$ of multi-target dependencies as complete, so that further dependencies may be identified and integrated in the future by examining other delta-oriented SPLs. The eight multi-target dependencies are defined as follows, where we focus for the definitions on three deltas for a better understanding, but multi-target dependencies allow for relations between more than three deltas in the same way:

- **Complete Alternative Dependency** (*calt*) – A delta $\delta \in \Delta_{DM}$ is *complete alternative dependent* to $\delta', \delta'' \in \Delta_{DM}$ denoted as $(n_\delta, calt, \{n_{\delta'}, n_{\delta''}\})$, iff δ is solely applicable for a variant $v \in \mathbb{V}$ either in combination with δ' or δ'' in a respective delta set Δ_v, but never with both deltas as δ' and δ'' are exclusive dependent:

$$(n_\delta, calt, \{n_{\delta'}, n_{\delta''}\}) \in Dep_{DM} \Leftrightarrow (n_{\delta'}, ex, n_{\delta''}) \wedge (n_\delta, pop, n_{\delta'}) \wedge (n_\delta, pop, n_{\delta''}) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'} \wedge \neg\varphi_{\delta''})$$

- **Partial Alternative Dependency** (*palt*) – A delta $\delta \in \Delta_{DM}$ is *partial alternative dependent* to $\delta', \delta'' \in \Delta_{DM}$ represented as $(n_\delta, palt, \{n_{\delta'}, n_{\delta''}\})$, iff δ is solely applicable for a variant $v \in \mathbb{V}$ either in combination with δ' or δ'' in a variant-specific delta set Δ_v. In contrast to the complete alternative dependency, where δ' and δ'' are not applicable without δ, the complete optional dependent δ'' is also applicable without δ in delta sets Δ_v:

$$(n_\delta, palt, \{n_{\delta'}, n_{\delta''}\}) \in Dep_{DM} \Leftrightarrow (n_{\delta'}, ex, n_{\delta''}) \wedge (n_\delta, pop, n_{\delta'}) \wedge (n_\delta, cop, n_{\delta''}) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'} \wedge \neg\varphi_{\delta''})$$

- **Partial Optional Alternative Dependency** (*popalt*) – A delta $\delta \in \Delta_{DM}$ is *partial optional alternative dependent* to $\delta', \delta'' \in \Delta_{DM}$ denoted as $(n_\delta, popalt, \{n_{\delta'}, n_{\delta''}\})$, iff δ is applicable for a variant $v \in \mathbb{V}$ either in combination with δ' or δ'' in a respective delta set Δ_v. In addition, the complete optional dependent δ'' is applicable without δ. In contrast to the partial alternative dependency, δ is also applicable without both deltas in delta sets Δ_v:

$$(n_\delta, popalt, \{n_{\delta'}, n_{\delta''}\}) \in Dep_{DM} \Leftrightarrow (n_{\delta'}, ex, n_{\delta''}) \wedge (n_\delta, pop, n_{\delta'}) \wedge (n_\delta, cop, n_{\delta''}) \wedge (\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'} \wedge \neg\varphi_{\delta''})$$

- **Complete Optional Alternative Dependency** (*copalt*) – A delta $\delta \in \Delta_{DM}$ is *complete optional alternative dependent* to $\delta', \delta'' \in \Delta_{DM}$ denoted as $(n_\delta, copalt, \{n_{\delta'}, n_{\delta''}\})$, iff δ is solely applicable for a variant $v \in \mathbb{V}$ either in combination with δ' or δ'' in a respective delta set Δ_v. In addition, δ' and δ'' are also independently applicable without δ:

$$(n_\delta, copalt, \{n_{\delta'}, n_{\delta''}\}) \in Dep_{DM} \Leftrightarrow (n_{\delta'}, ex, n_{\delta''}) \wedge (n_\delta, cop, n_{\delta'}) \wedge (n_\delta, cop, n_{\delta''}) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'} \wedge \neg\varphi_{\delta''})$$

- **Optional Alternative Dependency** (*opalt*) – A delta $\delta \in \Delta_{DM}$ is *optional alternative dependent* to $\delta', \delta'' \in \Delta_{DM}$ represented as $(n_\delta, opalt, \{n_{\delta'}, n_{\delta''}\})$, iff δ is applicable for a variant $v \in \mathbb{V}$ either (1) in combination with δ' or δ'', or (2) without both deltas in a respective delta set Δ_v. In addition, the complete optional dependent δ' and δ'' are individually applicable without δ, but not together due to the exclusive dependency between them:

$$(n_\delta, opalt, \{n_{\delta'}, n_{\delta''}\}) \in Dep_{DM} \Leftrightarrow (n_{\delta'}, ex, n_{\delta''}) \wedge (n_\delta, cop, n_{\delta'}) \wedge (n_\delta, cop, n_{\delta''}) \wedge (\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'} \wedge \neg\varphi_{\delta''})$$

- **Non-Exclusive Alternative Dependency** (*nexalt*) – A delta $\delta \in \Delta_{DM}$ is *non-exclusive alternative dependent* to $\delta', \delta'' \in \Delta_{DM}$ represented as $(n_\delta, nexalt, \{n_{\delta'}, n_{\delta''}\})$, iff δ is solely applicable for a variant $v \in \mathbb{V}$ either in combination with δ' or δ'' in a respective variant-specific delta set Δ_v. In contrast to the complete alternative dependency, where δ' and δ'' are not applicable together, both deltas are applicable together with δ due to the complete optional dependency between them:

$$(n_\delta, nexalt, \{n_{\delta'}, n_{\delta''}\}) \in Dep_{DM} \Leftrightarrow (n_{\delta'}, cop, n_{\delta''}) \wedge (n_\delta, pop, n_{\delta'}) \wedge (n_\delta, pop, n_{\delta''}) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'} \wedge \neg\varphi_{\delta''})$$

- **Non-Exclusive Partial Alternative Dependency** (*nexpalt*) – A delta $\delta \in \Delta_{DM}$ is *non-exclusive partial alternative dependent* to $\delta', \delta'' \in \Delta_{DM}$ denoted as $(n_\delta, nexpalt, \{n_{\delta'}, n_{\delta''}\})$, iff δ is applicable for a variant $v \in \mathbb{V}$ either (1) in combination with δ' or δ'', or (2) with both deltas in a delta set Δ_v. In addition, the complete optional dependent δ'' is applicable without δ:

$$(n_\delta, nexpalt, \{n_{\delta'}, n_{\delta''}\}) \in Dep_{DM} \Leftrightarrow (n_{\delta'}, cop, n_{\delta''}) \wedge (n_\delta, pop, n_{\delta'}) \wedge (n_\delta, cop, n_{\delta''}) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'} \wedge \neg\varphi_{\delta''})$$

- **Implication Dependency** (*imp*) – A delta $\delta \in \Delta_{DM}$ is *implication dependent* to $\delta', \delta'' \in \Delta_{DM}$ represented as $(n_\delta, imp, \{n_{\delta'}, n_{\delta''}\})$, iff δ' and δ'' are partial optional dependent with δ and both deltas are not applicable together without δ in a variant-specific delta set Δ_v:

$$(n_\delta, imp, \{n_{\delta'}, n_{\delta''}\}) \in Dep_{DM} \Leftrightarrow (n_{\delta'}, pop, n_\delta) \wedge (n_{\delta''}, pop, n_\delta) \wedge (\neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \neg\varphi_\delta \wedge \varphi_{\delta'} \wedge \varphi_{\delta''})$$

We perform the dependency analysis solely based on the application conditions of deltas which is sufficient to reason about the possible combinations, i.e., dependencies, of deltas. There also exist dependencies between deltas on state machine level, e.g., one delta adds the source and target state of a transition which is added by another delta. However, those low-level dependencies between deltas are also captured by their application conditions as the application conditions have to be defined such that both deltas can be applied together. In case the low-level dependencies are not represented by the defined application conditions, the delta model is not valid and should be corrected [CHS15]. As described in Sect. 3.2, we assume a valid delta model to be given such that the dependencies between deltas on the state machine level are also captured by their application conditions.

We compute a delta dependency graph DG^θ_{DM} for a delta set Δ_{DM_θ} of the delta model DM_θ of an SPL version θ under consideration as follows, where the corresponding algorithm is shown in Alg. 4.14 in pseudo code. As input for the delta dependency graph generation, we use the delta set Δ_{DM_θ} and the feature model fm_θ of the SPL version θ. As result, we return the generated delta dependency graph DG^θ_{DM} capturing the information about the relations between deltas by means of their potential combination for variant-specific delta sets.

Algorithm 4.14.: Delta Dependency Graph Generation

Input: Delta Set Δ_{DM} and Feature Model fm
Output: Delta Dependency Graph DG_{DM}

```
1 Function buildDeltaDepGraph
2     DG_DM := initDeltaNodes(Δ_DM);
3     DG_DM := determineSingleDeltaDeps(Δ_DM, DG_DM, fm);
4     DG_DM := determineMutliDeltaDeps(Δ_DM, DG_DM, fm);
5 return DG_DM;
```

The generation includes three main steps to obtain the delta dependency graph DG^θ_{DM}. First, we initialize the set of delta nodes N_{DM} by creating for each delta $\delta \in \Delta_{DM_\theta}$ a delta node $n_\delta \in N_{DM}$ (cf. Line 2). Second, we determine the single-target dependencies that exist between state machine deltas $\delta, \delta' \in \Delta_{DM_\theta}$ of the delta model DM_θ (cf. Line 3). Therefore, we check each delta $\delta \in \Delta_{DM_\theta}$

with the remaining deltas $\delta' \in \Delta_{DM_\theta}$ comprised in the delta set Δ_{DM_θ} by examining the satisfiability of the potential combination of their application conditions w.r.t. the given feature model *fm*. For the derivation of a single-target dependency, we have to consider three cases and their satisfiability:

1. Can both deltas be applied together? –

$$\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \varphi_{\delta'} \text{ or } \neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \varphi_{\delta'}$$

2. Can delta δ be applied without delta δ'? –

$$\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'} \text{ or } \neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \varphi_\delta \wedge \neg\varphi_{\delta'}$$

3. Can delta δ' be applied without delta δ? –

$$\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \neg\varphi_\delta \wedge \varphi_{\delta'} \text{ or } \neg\exists v \in \mathbb{V} : [\![F_v]\!] \models [\![fm]\!]_{\mathbb{B}} \wedge \neg\varphi_\delta \wedge \varphi_{\delta'}$$

Depending on the outcome of the three satisfiability checks exactly one of the four defined single-target dependency types is applicable. For instance, if the first and second case is satisfiable, but the third case is not, we identify a partial optional dependency between delta δ and delta δ', i.e., δ' requires δ to be selected for a variant-specific delta set. Each detected dependency is added to the delta dependency graph as dependency edge $(n_\delta, l^{single}_{\Delta_{Dep}}, n_{\delta'})$ with $l^{single}_{\Delta_{Dep}} \in \mathcal{L}^{single}_{\Delta_{Dep}}$ between the delta nodes $n_\delta, n_{\delta'} \in N_{DM}$ of the dependent deltas δ and δ'. As third and final step, we exploit the determined single-target dependencies to derive the potential existing multi-target dependencies (cf. Line 4). For each delta node $n_\delta \in N_{DM}$ and, thus, for each delta $\delta \in \Delta_{DM_\theta}$, we examine its existing single-target dependencies and check for the connected delta nodes $n_{\delta'} \in N_{DM}$ of deltas $\delta' \in \Delta_{DM_\theta}$ whether the prerequisites for a multi-target dependency are given. If a multi-target dependency can be established, we add an dependency edge $(n_\delta, l^{multi}_{\Delta_{Dep}}, \{n_{\delta'}, n_{\delta''}, \ldots\})$ with $l^{multi}_{\Delta_{Dep}} \in \mathcal{L}^{multi}_{\Delta_{Dep}}$ between the respective delta nodes of the dependent deltas. In the end, we return the generated delta dependency graph DG^θ_{DM} and afterwards exploit its captured information about the relations between deltas to create a variant tree that facilitates an automated derivation of variant-specific delta sets.

Example 4.5: Delta Dependency Graph Generation

Consider the sample delta dependency graph $DG^{\theta_0}_{DM}$ of delta model DM_{θ_0} from Ex. 3.3 for SPL Version θ_0 of our running example shown in Fig. 4.8. For a better representation, we split up the graph in two parts, where in Fig. 4.8a the single-target dependencies are depicted and in Fig. 4.8b the multi-target dependencies. We generate this graph by applying the three steps of Alg. 4.14. First, we create for each delta $\delta_j \in \Delta_{DM_{\theta_0}}$ of DM_{θ_0} a respective delta node in the graph. Second, we determine the set of existing single-target dependencies, where we obtain the dependencies $(\delta_1, man, \delta_2)$, $(\delta_2, man, \delta_1)$, $(\delta_1, pop, \delta_3)$, $(\delta_1, cop, \delta_4)$, $(\delta_4, cop, \delta_1)$, $(\delta_1, pop, \delta_5)$, $(\delta_2, pop, \delta_3)$, $(\delta_2, pop, \delta_5)$, $(\delta_2, cop, \delta_4)$, $(\delta_4, cop, \delta_2)$, (δ_3, ex, δ_4), (δ_4, ex, δ_3), (δ_3, ex, δ_5), (δ_5, ex, δ_3), and $(\delta_4, pop, \delta_5)$. For instance, we identified a mandatory dependency $(\delta_1, man, \delta_2)$ between δ_1 and δ_2 as the three satisfiability checks result in

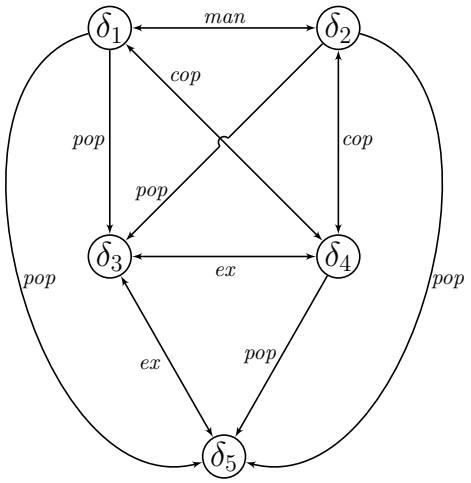
1. $\exists v \in \mathbb{V}_{\theta_0} : [\![F_v]\!] \models [\![fm_{\theta_0}]\!]_{\mathbb{B}} \wedge f_1 \wedge f_1$ – δ_1 and δ_2 are applicable together.
2. $\neg\exists v \in \mathbb{V}_{\theta_0} : [\![F_v]\!] \models [\![fm_{\theta_0}]\!]_{\mathbb{B}} \wedge f_1 \wedge \neg f_1$ – δ_1 is not applicable without δ_2
3. $\neg\exists v \in \mathbb{V}_{\theta_0} : [\![F_v]\!] \models [\![fm_{\theta_0}]\!]_{\mathbb{B}} \wedge \neg f_1 \wedge f_1$ – δ_2 is not applicable without δ_1

For the representation of fm_{θ_0} as propositional formula via $[\![fm_{\theta_0}]\!]_{\mathbb{B}}$, we refer to Ex. 2.1 for the respective definition. Third, we determine the multi-target dependencies, where we obtain the dependencies $(\delta_1, palt, \{\delta_3, \delta_4\})$, $(\delta_1, calt, \{\delta_3, \delta_5\})$, $(\delta_2, calt, \{\delta_3, \delta_5\})$, $(\delta_2, palt, \{\delta_3, \delta_4\})$, $(\delta_5, imp, \{\delta_1, \delta_4\})$, and $(\delta_5, imp, \{\delta_2, \delta_4\})$. For example, we identified a complete alternative dependency $(\delta_1, calt, \{\delta_3, \delta_5\})$ between δ_1, δ_3, and δ_5 as the respective prerequisites hold:

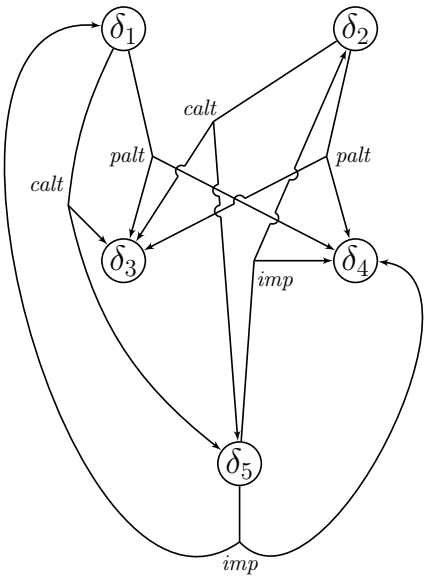
$$(\delta_3, ex, \delta_5) \wedge (\delta_1, pop, \delta_3) \wedge (\delta_1, pop, \delta_5) \wedge$$
$$(\neg\exists v \in \mathbb{V}_{\theta_0} : [\![F_v]\!] \models [\![fm_{\theta_0}]\!]_{\mathbb{B}} \wedge f_1 \wedge \neg(f_2 \wedge f_1) \wedge \neg(f_3 \wedge f_1))$$

In our running example, there exist solely multi-target dependencies w.r.t. three deltas. The resulting delta dependency graph $DG^{\theta_0}_{DM}$ is defined by $N_{DM} = \{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5\}$ and
$Dep_{DM} = \{(\delta_1, man, \delta_2), (\delta_2, man, \delta_1), (\delta_1, pop, \delta_3), (\delta_1, cop, \delta_4), (\delta_4, cop, \delta_1), (\delta_1, pop, \delta_5),$
$(\delta_2, pop, \delta_3), (\delta_2, pop, \delta_5), (\delta_2, cop, \delta_4), (\delta_4, cop, \delta_2), (\delta_3, ex, \delta_4), (\delta_4, ex, \delta_3), (\delta_3, ex, \delta_5),$
$(\delta_5, ex, \delta_3), (\delta_4, pop, \delta_5), (\delta_1, palt, \{\delta_3, \delta_4\}), (\delta_1, calt, \{\delta_3, \delta_5\}), (\delta_2, calt, \{\delta_3, \delta_5\}),$
$(\delta_2, palt, \{\delta_3, \delta_4\}), (\delta_5, imp, \{\delta_1, \delta_4\}), (\delta_5, imp, \{\delta_2, \delta_4\})\}$.

Variant Tree Generation. For the delta set derivation, we propose a *variant tree* capturing all derivable delta sets in a compact way as variant tree paths. Furthermore, it records for each non-derivable delta set the respective set of restrictions specified by unfulfilled delta dependencies of the delta dependency graph DG_{DM}. Based on this representation, we ensure the traceability how as well as which delta sets may change due to the application of a higher-order delta. A variant tree represents an adaptation of binary trees such that the tree starts in an empty root node and each tree node has at most two child nodes distinguished as left and right child. The distinct hierarchy levels of a



(a) Sample Delta Dependency $DG^{\theta_0}_{DM}$ Graph with Single-Target Dependencies



(b) Sample Delta Dependency $DG^{\theta_0}_{DM}$ Graph with Multi-Target Dependencies

Figure 4.8: Sample Delta Dependency Graph $DG^{\theta_0}_{DM}$ of Delta Model DM_{θ_0} for SPL Version θ_0

variant tree which are spanned based on the parent-child relation correspond to deltas $\delta \in \Delta_{DM}$ of a delta model DM and their containment in a potential variant-specific delta set Δ_v. In case a delta tree node is a left child, the respective delta is part of a variant-specific delta set which is derivable by the path from a leaf tree node to the root node, whereas the delta is not part of a variant-specific delta set if its delta tree node is a right child. For the decision whether delta tree nodes for a delta are integrated in the variant tree as left, right, or as left and right child of a respective parent delta tree node, we take the delta dependencies Dep_{DM} of the generated delta dependency graph DG_{DM} into account. Therefore, we check for each preliminary tree path starting in the root node, whether the delta tree node to be added has restrictions to its predecessor tree nodes by means of dependencies between their mapped deltas. For instance, a mandatory dependency implies the integration as left child and hinders the integration as right child, whereas an exclusive dependency prevents from the integration as left child. If a restriction is found, we integrate a restricted delta tree node and relate the node to the restriction caused by delta dependencies. Otherwise, i.e., no restriction is identified, the tree nodes of a delta are added both as left and right child to the current path.

Definition 4.10: Variant Tree

Let $\mathcal{VT}$ be the universe of all variant trees defined over the universe of all delta models $\mathcal{DM}$. A *variant tree* $VT = (N_{VT}, \top, \prec_L, \prec_R, Dep_{DM}, \alpha, \Delta_{DM}, \lambda)$ is an 8-tuple, where

- $N_{VT} = \{\top\} \cup N_\Delta \cup N_\Delta^\otimes$ is a finite set of tree nodes with
  - $\{\top\}$ is the set solely comprising the empty root node,
  - $N_\Delta = \{n_1, \ldots, n_m\}$ is a finite set of delta tree nodes, and
  - $N_\Delta^\otimes = \{n_1^\otimes, \ldots, n_l^\otimes\}$ is a finite set of restricted delta tree nodes,
- $\top$ is the empty root node,
- $\prec_L \colon N_{VT} \to N_\Delta \cup N_\Delta^\otimes \cup \{\otimes\} \cup \{\bot\}$ is the left-child hierarchy function mapping a delta tree node $n \in N_{VT}$ either (1) to its left child $n' \in N_\Delta$, (2) to a restricted delta tree node $n^\otimes \in N_\Delta^\otimes$ representing that no subtree derivation is required due to restrictions, (3) to the special symbol $\otimes$ denoting that there is no left child due to restrictions, or (4) to the special symbol $\bot$ representing that the delta tree node n is a leaf of the variant tree,
- $\prec_R \colon N_{VT} \to N_\Delta \cup N_\Delta^\otimes \cup \{\otimes\} \cup \{\bot\}$ is the right-child hierarchy function mapping a tree node $n \in N_{VT}$ either (1) to its right child $n' \in N_\Delta$, (2) to a restricted tree node $n^\otimes \in N_\Delta^\otimes$ representing that no subtree derivation is required due to restrictions, (3) to the special symbol $\otimes$ denoting that there is no right child due to restrictions, or (4) to the special symbol $\bot$ representing that the tree node n is a leaf of the variant tree,
- Dep_{DM} is the set of delta dependencies,
- $\alpha : N_{VT} \to \mathcal{P}(Dep_{DM})$ is the restriction mapping function,
- Δ_{DM} is the finite delta set of the delta model DM, and
- $\lambda : N_\Delta \cup N_\Delta^\otimes \to \Delta_{DM}$ is the labeling function mapping a (restricted) delta tree node to its respective delta $\delta \in \Delta_{DM}$.

A variant tree is well-formed facilitating the derivation of variant-specific delta sets if the tree ensures some properties. For the definition of those properties, we use the *hierarchy function* $\prec \colon \mathcal{VT} \times \mathbb{N}_0 \to \mathcal{P}(N_{VT})$ which returns for a given variant tree $VT \in \mathcal{VT}$ and a hierarchy level $level \in \mathbb{N}_0$ the set of respective tree nodes such that

$$\prec(VT, level) = \begin{cases} \{\top\} & \text{if } level = 0 \\ N_{VT}^{level} \subset N_{VT} & \text{if } 0 < level \leq |\Delta_{DM}| \\ N_{VT}^{|\Delta_{DM}|} \subset N_{VT} & \text{if } |\Delta_{DM}| < level \end{cases}$$

holds. The well-formedness properties are as follows:

- The root node $\top$ has no parent tree node which follows from the definition of the child hierarchy functions $\prec_L$ and $\prec_R$
- The root node $\top$ has no restrictions: $\alpha(\top) = \emptyset$
- Every (restricted) delta tree node of the same hierarchy level is mapped to the same delta $\delta \in \Delta_{DM}$: $\forall n \in \prec(VT, level) = N_{VT}^{level}, level > 0 : \lambda(n) = \delta$
- All delta tree nodes of the last hierarchy level (leaf nodes) map to the special symbol $\perp$ as left and right child: $\forall n \in \prec(VT, |\Delta_{DM}|) = N_{VT}^{|\Delta_{DM}|} : (\prec_L(n) = \perp) \wedge (\prec_R(n) = \perp)$
- All restricted delta tree nodes and only these nodes map to the special symbol $\otimes$ as left and right child: $(\forall n \in N_{\Delta}^{\otimes} : (\prec_L(n) = \otimes) \wedge (\prec_R(n) = \otimes)) \wedge (\neg\exists n' \in N_{VT} \setminus N_{\Delta}^{\otimes} : (\prec_L(n') = \otimes) \vee (\prec_R(n') = \otimes))$

In a variant tree, each hierarchy level is mapped to a certain delta $\delta \in \Delta_{DM}$ such that the order of deltas to be integrated into a variant tree defines its structure. Depending on the delta order, the variant tree structure will differ. However, the result of structural different variant trees for the same delta set Δ_{DM} and delta dependencies Dep_{DM} will be the same by means of the same set of variant-specific delta sets which are derivable as variant tree paths. In this thesis, we take the order in which deltas are created or rather the order in which deltas are contained in the set Δ_{DM} into account and do not require a specific order for the variant tree creation.

As already mentioned, a delta set Δ_v of a variant $v \in \mathbb{V}$ is defined by a complete path from a leaf to the root node by incorporating whether a delta tree node is a left or right child. In case a node is a left child, the respective delta is comprised in a derivable delta set. In contrast, a right child node denotes that the mapped delta is not contained in such a set.

Definition 4.11: Variant-Tree Path

Let $\Delta_{\rho_{VT}}$ be the set of all derivable variant-tree paths of a variant tree VT. The functions $\prec_L^{-1}$ and $\prec_R^{-1}$ denote the inverse left-child and right-child hierarchy functions such that the parent of a delta tree node is returned. A *variant-tree path* $\rho_{VT} = ((n_1, \ldots, n_m), \lambda_{\prec}) \in \Delta_{\rho_{VT}}$ is defined as tuple, where

- $(n_1, \ldots, n_m) \in N_{\Delta}^{*}$ is sequence of delta tree nodes such that
  - $\prec_L(n_1) = \perp \wedge \prec_R(n_1) = \perp$, i.e., the first tree node n_1 is a leaf of the variant tree,
  - $\prec_L^{-1}(n_m) = \top \vee \prec_R^{-1}(n_m) = \top$, i.e., the last tree node n_m is a left or right child node of the root node of the variant tree, and
  - $\forall n_i, 1 \leq i < m : \prec_L^{-1}(n_i) = n_{i+1} \vee \prec_R^{-1}(n_i) = n_{i+1}$, i.e., a delta tree node n_i is either a right or left child of the next delta tree node n_{i+1} of the sequence representing the variant-tree path,

  holds, and

- $\lambda_{\prec} : N_{\Delta} \rightarrow \{L, R\}$ is a labeling function such that

$$\forall n_i, 1 \leq i < m : \lambda_{\prec}(n_i) = \begin{cases} L & \text{if } \prec_L (n_{i+1}) = n_i \\ R & \text{if } \prec_R (n_{i+1}) = n_i \end{cases}$$

  holds, i.e., the function provides for each delta tree node of the path either an L or an R as marker indicating whether a tree node is a left or right child node, respectively.

Based on those definitions, a variant tree captures all variants $v \in \mathbb{V}_{\theta}$ of an SPL version θ in terms of their delta sets Δ_v represented as variant-tree paths $\rho^v_{VT} \in \Delta^{\theta}_{\rho_{VT}}$ in a compact way which is exploited to reason about the impact of the application of a higher-order delta on the set of variants as described in Sect. 4.2.3. In contrast to this benefit, a common limitation of binary trees and, thus, also of our proposed variant tree is its exponential growth in the number of tree nodes w.r.t. the number of deltas. Depending on the number of deltas $\delta \in \Delta_{DM}$, the height of a variant tree is defined as $h = |\Delta_{DM}| + 1$. Hence, a variant tree comprises a total number of $2^h - 1$ tree nodes with $2^{|\Delta_{DM}|}$ leaf tree nodes in the worst case. However, the worst case denoting a full variant tree only occurs if all deltas are complete optional to each other, i.e., every delta is combinable with every other delta in variant-specific delta sets, which denotes a very special scenario. In the average case, there exist restrictions between deltas regarding their combination such that they are not applicable together for a variant. Thus, the number of tree nodes is reduced as respective subtrees are not computed if the integration of a delta node into the variant tree is restricted. Furthermore, the number of leaf tree nodes is predefined by the number of variants of the SPL under consideration, where $|\mathbb{V}| \leq 2^{|\Delta_{DM}|}$ holds, such that the total number of tree nodes strongly depends on the number of variants. Still, the size of a variant tree may be a limitation for larger SPLs, but the huge number of variants of larger SPLs is a general challenge in SPLE [PBvdL05] and is not specific for our variant trees. To cope with this challenge, sampling strategies are applied to determine a representative subset of variants [VAT+18], e.g., to perform testing [JHF12; AKT+16a]. We are able to exploit sampling such that the delta dependency analysis and variant tree generation is solely applied to the determined subset of variants which reduces the size of a variant tree w.r.t. the total number of tree nodes. In addition, a respective tool support for the delta set derivation with a suitable data structure to capture variant trees facilitates the reduction of the increasing memory footprint.

We generate a variant tree VT for a delta set $\Delta_{DM_{\theta}}$ of the delta model DM_{θ} of an SPL version θ based on a given delta dependency graph DG^{θ}_{DM} as follows, where the corresponding algorithm is shown in Alg. 4.15 in pseudo code. The pseudo code algorithms of the auxiliary functions `addChild`, `getPath`, and `checkDepForRestriction` are shown in Alg. 4.16, Alg. 4.17, and Alg. 4.18, respectively.

We start the generation by (1) initializing the variant tree with the addition of the root node $\top$ (cf. Line 2), and (2) setting the hierarchy level variable *level* to 0. We exploit the variable to determine the set of nodes of a hierarchy level using the hierarchy function $\prec$. After this initialization phase, we iterate over the set of deltas $\delta \in \Delta_{DM}$ of the delta model DM and check how a delta is integrated in the intermediate variant tree. In case the tree is empty (cf. Line 5), i.e., it solely comprises the root node $\top$, we integrate the first delta (cf. Line 6) by adding tree nodes as left and right child of the root node specified by the parameter `LEFT_RIGHT` $\in$ {`LEFT_RIGHT`, `LEFT`, `RIGHT`} of function *addChild*, where further $\emptyset$ denotes that no restrictions exist. As shown in Alg. 4.16, we create two new delta

Algorithm 4.15.: Variant Tree Computation

Input: Delta Dependency Graph DG_{DM} and Delta Set Δ_{DM}
Output: Variant Tree VT

```
1  Function buildVariantTree
2      VT := ({⊤}, ⊤, ≺_L:≺_L (⊤) = ⊥, ≺_R:≺_R (⊤) = ⊥, Dep_DM, α : α(⊤) = ∅, Δ_DM, λ);
3      level := 0;
4      forall δ ∈ Δ_DM do
5          if N_VT == {⊤} then
6              VT := addChild(VT, ⊤, LEFT_RIGHT, δ, ∅);
7          else
8              N_Δ^level := getLeafs(VT, level);
9              forall n_δ' ∈ N_Δ^level do
10                 Dep_DM^δ,L := ∅;
11                 Dep_DM^δ,R := ∅;
12                 ρ_VT := getPath(VT, n_δ');
13                 Dep_DM^δ,L := checkDepForRestriction(VT, DG_DM, ρ_VT, δ, LEFT);
14                 Dep_DM^δ,R := checkDepForRestriction(VT, DG_DM, ρ_VT, δ, RIGHT);
15                 if Dep_DM^δ,L == ∅ ∧ Dep_DM^δ,R == ∅ then
16                     VT := addChild(VT, n_δ', LEFT_RIGHT, δ, ∅);
17                 else if Dep_DM^δ,L == ∅ ∧ Dep_DM^δ,R ≠ ∅ then
18                     VT := addChild(VT, n_δ', LEFT, δ, Dep_DM^δ);
19                 else if Dep_DM^δ,L ≠ ∅ ∧ Dep_DM^δ,R == ∅ then
20                     VT := addChild(VT, n_δ', RIGHT, δ, Dep_DM^δ);
21         level++;
22     return VT;
```

nodes $n_\delta^{n_{parent},L}$ and $n_\delta^{n_{parent},R}$ and add them to the set N_{VT} of delta nodes. Afterwards, we update the mapping between the parent node n_{parent} and its left as well as right child to the respective new delta nodes and also define the mappping of the new delta nodes to the special leaf symbol $\perp$. As we integrate the delta as left and right child, there exist no restrictions such that we define the value of α as $\emptyset$. As last step, both new delta tree nodes are mapped to the delta δ to be integrated.

For the remaining, not yet incorporated deltas, we first determine the set of delta tree nodes comprised in the last hierarchy level which was integrated denoting the current leaf nodes of the variant tree (cf. Line 8). The function `getLeafs` uses internally the hierarchy function $\prec$ and neglects restricted delta tree nodes such that the integration of new delta tree nodes is solely applied to non-restricted delta tree nodes in the following steps. We iterate over the leaf nodes and determine, for the current delta under consideration, how to be integrated in the variant tree w.r.t. restrictions existing to deltas already comprised in the variant tree (cf. Line 9ff). To record potential restrictions for the current delta to be integrated by means of delta dependencies, we use the set $Dep_{DM}^{\delta,L}$ for restricting the addition as left child and $Dep_{DM}^{\delta,R}$ for restricting the addition as right child, accordingly.

For the determination of restrictions, we follow the preliminary variant tree path starting in the current leaf node n_δ up to the root node and check for delta dependencies from the delta dependency graph DG_{DM} which potentially restrict the addition of a delta tree node as left or right child.

Algorithm 4.16.: Function `addChild`

Input: Intermediate Variant Tree VT, Parent Tree Node n_{parent}, Parameter for Node Integration $param$, Delta to be integrated δ, Set of Restricting Delta Dependencies Dep^{δ}_{DM}

Output: Updated Variant Tree VT

1 **Function** *addChild*
2 **if** $param == LEFT\_RIGHT$ **then**
3 $N_{VT} := N_{VT} \cup \{n_{\delta}^{n_{parent},L}, n_{\delta}^{n_{parent},R}\}$;
4 $\prec_L (n_{parent}) := n_{\delta}^{n_{parent},L}$;
5 $\prec_R (n_{parent}) := n_{\delta}^{n_{parent},R}$;
6 $\prec_L (n_{\delta}^{n_{parent},L}) := \bot$;
7 $\prec_R (n_{\delta}^{n_{parent},R}) := \bot$;
8 $\alpha(n_{\delta}^{n_{parent},L}) := \emptyset$;
9 $\alpha(n_{\delta}^{n_{parent},R}) := \emptyset$;
10 $\lambda(n_{\delta}^{n_{parent},L}) := \delta$;
11 $\lambda(n_{\delta}^{n_{parent},R}) := \delta$;
12 **else if** $param == LEFT$ **then**
13 $N_{VT} := N_{VT} \cup \{n_{\delta}^{n_{parent},L}, n_{\delta}^{\otimes,n_{parent},R}\}$;
14 $\prec_L (n_{parent}) := n_{\delta}^{n_{parent},L}$;
15 $\prec_R (n_{parent}) := n_{\delta}^{\otimes,n_{parent},R}$;
16 $\prec_L (n_{\delta}^{n_{parent},L}) := \bot$;
17 $\prec_R (n_{\delta}^{\otimes,n_{parent},R}) := \otimes$;
18 $\alpha(n_{\delta}^{n_{parent},L}) := \emptyset$;
19 $\alpha(n_{\delta}^{\otimes,n_{parent},R}) := Dep^{\delta}_{DM}$;
20 $\lambda(n_{\delta}^{n_{parent},L}) := \delta$;
21 $\lambda(n_{\delta}^{\otimes,n_{parent},R}) := \delta$;
22 **else**
23 $N_{VT} := N_{VT} \cup \{n_{\delta}^{n_{\otimes},parent,L}, n_{\delta}^{n_{parent},R}\}$;
24 $\prec_L (n_{parent}) := n_{\delta}^{\otimes,n_{parent},L}$;
25 $\prec_R (n_{parent}) := n_{\delta}^{n_{parent},R}$;
26 $\prec_L (n_{\delta}^{\otimes,n_{parent},L}) := \otimes$;
27 $\prec_R (n_{\delta}^{n_{parent},R}) := \bot$;
28 $\alpha(n_{\delta}^{\otimes,n_{parent},L}) := Dep^{\delta}_{DM}$;
29 $\alpha(n_{\delta}^{n_{parent},R}) := \emptyset$;
30 $\lambda(n_{\delta}^{\otimes,n_{parent},L}) := \delta$;
31 $\lambda(n_{\delta}^{n_{parent},R}) := \delta$;
32 **return** VT;

Hence, we first derive the preliminary variant tree path via the function `getPath` (cf. Line 12). As shown in Alg. 4.17, we start the derivation by adding the leaf node to the tree path and initialize $\lambda_{\prec}$ by incorporating the hierarchy relation to its parent tree node. In case the parent node is not the

Algorithm 4.17.: Function `getPath`

Input: Variant Tree VT, Leaf Tree Node n_δ^{leaf}
Output: Variant-Tree Path ρ_{VT}

```
1  Function getPath
2      n_δ^child := n_δ^leaf;
3      n_δ^parent := getParent(VT, n_δ^child);
4      ρ_VT := ((n_δ^child), λ_≺);
5      if n_δ^child ==≺_L (n_δ^parent) then
6          λ_≺(n_δ^child) := L;
7      else
8          λ_≺(n_δ^child) := R;
9      while n_δ^parent ≠ ⊤ do
10         n_δ^child := n_δ^parent;
11         n_δ^parent := getParent(VT, n_δ^child);
12         ρ_VT := ((..., n_δ^child), λ_≺);
13         if n_child ==≺_L (n_δ^parent) then
14             λ_≺(n_δ^child) := L;
15         else
16             λ_≺(n_δ^child) := R;
17 return ρ_VT;
```

root tree node $\top$, we traverse the path up to the root node, where we repeat the steps of adding the current child node n_δ^{child} to the tree path and updating the function $\lambda_\prec$ by incorporating the hierarchy relation to the current parent tree node n_δ^{parent}.

The tree path ρ_{VT} is afterwards used to determine the potential restrictions for the integration of the delta δ as left child (cf. Line 13) and right child (cf. Line 14) captured as delta dependencies in the delta dependency graph DG_{DM} via the function `checkDepForRestriction`. As depicted in Alg. 4.18, we first determine the sets of left N_L and right N_R delta dependency graph nodes of deltas contained in the preliminary variant tree path ρ_{VT}. Both sets are required to evaluate whether existing dependencies restrict the addition of the delta into the tree. As second step, we get the respective delta node n_δ for the delta δ to be integrated from the dependency graph DG_{DM} and also determine all dependencies Dep_{DM}^δ captured in DG_{DM} the delta node n_δ is involved in. By iterating over the determined set Dep_{DM}^δ, we evaluate for each dependency $dep \in Dep_{DM}^\delta$ whether the dependency is valid w.r.t. the sets of left N_L and right N_R delta dependency graph nodes of deltas already part of the variant tree as well as the delta node n_δ of the delta to be integrated. The parameter $param \in \{\texttt{LEFT}, \texttt{RIGHT}\}$ controls in this process that the delta is handled either as part of a derivable delta set (`LEFT`) or not (`RIGHT`). For instance, assume there exist a mandatory dependency between a delta δ' which is already part of the variant tree and the delta δ to be integrated. In case the delta tree node $n_{\delta'}$ of the already integrated delta δ' is contained in the current path ρ_{VT} under consideration as right child, i.e., the respective delta dependency graph node $n_{\delta'}$ is part of N_R, and the delta δ should be integrated as left child in the current path, the mandatory dependency cannot be fulfilled and is, therefore, invalid representing a restriction for the delta integration. For the

Algorithm 4.18.: Function `checkDepForRestriction`

Input: Variant Tree VT, Delta Dependency Graph DG_{DM}, Variant-Tree Path ρ_{VT}, Delta to be integrated δ, Parameter for Node Integration *param*

Output: Set of Restricting Delta Dependencies Dep^{δ}_{DM}

```
1  Function checkDepForRestriction
2      N_L := ∅;
3      N_R := ∅;
4      forall n ∈ ρ_VT do
5          n_DG_DM := getNode(DG_DM, λ(n));
6          if λ_≺(n) = L then
7              N_L := N_L ∪ {n_DG_DM};
8          else
9              N_R := N_R ∪ {n_DG_DM}
10     n_δ := getNode(DG_DM, δ);
11     Dep^δ_DM := getDependencies(DG_DM, n_δ);
12     forall dep ∈ Dep^δ_DM do
13         if param == LEFT then
14             if !valid(dep, N_L, N_R, n_δ) then
15                 Dep^δ_DM := Dep^δ_DM ∪ {dep};
16         else
17             if !valid(dep, N_L, N_R, ¬n_δ) then
18                 Dep^δ_DM := Dep^δ_DM ∪ {dep};
19 return Dep^δ_DM;
```

evaluation whether a dependency is valid or not, we refer to the definition of the delta dependencies as described above. Please note, we handle a dependency which cannot be evaluated during the current execution of `checkDepForRestriction` as valid. Such a situation occurs if not all deltas required for the evaluation are already part of the variant tree.

After we finished the derivation of potential restrictions, we check whether the sets $Dep^{\delta,L}_{DM}$ and $Dep^{\delta,R}_{DM}$ are empty or not and add respective delta tree nodes to the variant tree. If both sets are empty, we add a left and right child tree node and label them with the delta δ (cf. Line 16). In case $Dep^{\delta,R}_{DM}$ is not empty, we add a delta tree node solely as left child (cf. Line 18). Otherwise, we add a delta tree node solely as right child node. As last step of the iteration, we increment the hierarchy level variable and start the next iteration until all deltas $\delta \in \Delta_{DM}$ of the delta model DM are integrated.

The algorithm for the variant tree computation terminates as the set of deltas to be integrated is finite. In the end, we obtain a variant tree VT for a delta set Δ_{DM_θ} of the delta model DM_θ of an SPL version θ under consideration that captures all variants $v \in \mathbb{V}_\theta$ in terms of their delta sets Δ_v represented as variant-tree paths $\rho^v_{VT} \in \Delta^{\theta}_{\rho_{VT}}$ in a compact way.

Example 4.6: Delta Set Derivation

Consider the sample variant tree VT_{θ_0} shown in Fig. 4.9 for the delta model DM_{θ_0} of SPL Version θ_0 of our running example described in Ex. 3.3. For the derivation of the restrictions,

we use the generated delta dependency graph $DG^{\theta_0}_{DM}$ from Ex. 4.5 depicted in Fig. 4.8. The variant tree VT_{θ_0} allows for the derivation of the four paths

- $\rho^{v_{core}}_{VT} = ((n^R_{\delta_5}, n^R_{\delta_4}, n^R_{\delta_3}, n^R_{\delta_2}, n^R_{\delta_1}), \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = \lambda_{\prec}(n_{\delta_3}) = \lambda_{\prec}(n_2) = \lambda_{\prec}(n_{\delta_1}) = R)$,
- $\rho^{v_1}_{VT} = ((n^R_{\delta_5}, n^R_{\delta_4}, n^L_{\delta_3}, n^L_{\delta_2}, n^L_{\delta_1}), \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = R; \lambda_{\prec}(n_{\delta_3}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta_1}) = L)$,
- $\rho^{v_2}_{VT} = ((n^R_{\delta_5}, n^L_{\delta_4}, n^R_{\delta_3}, n^R_{\delta_2}, n^R_{\delta_1}), \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_3}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta_1}) = R; \lambda_{\prec}(n_{\delta_4}) = L)$, and
- $\rho^{v_3}_{VT} = ((n^L_{\delta_5}, n^L_{\delta_4}, n^R_{\delta_3}, n^L_{\delta_2}, n^L_{\delta_1}), \lambda_{\prec}(n_{\delta_3}) = R; \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta_1}) = L)$.

The four variant tree paths correspond to the four variant-specific delta sets $\Delta_{v_{core}}$, Δ_{v_1}, Δ_{v_2}, Δ_{v_3} which are also derivable by evaluating and selecting a delta from $\Delta_{DM_{\theta_0}}$ if its application condition is satisfied by the feature configuration $F_{v_i} \in F_{\mathbb{V}_{\theta_0}}$, i.e., $[\![F_{v_i}]\!] \models [\![fm_{\theta_0}]\!]_{\mathbb{B}} \wedge \varphi_\delta$.

In Fig. 4.9, we also provide, for each restricted delta node represented by $\otimes$, its restricting delta dependencies. For instance, we cannot add a left-child node for the delta δ_4 under the parent node for delta δ_3 as the parent node is itself a left-child tree node and there exist an exclusive dependency between δ_3 and δ_4. Furthermore, as we can see, a delta tree node is not necessarily restricted by solely one dependency, but rather based on a set of dependencies.

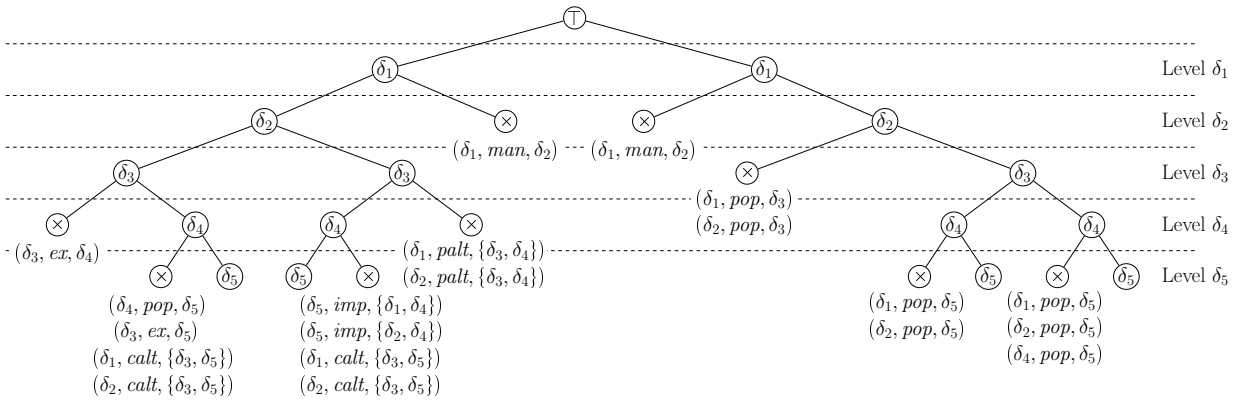


Figure 4.9: Sample Variant Tree VT_{θ_0} for the Delta Model DM_{θ_0} of SPL Version θ_0

In Tab. 4.4, we summarize the list of symbols used for the definition of the delta set derivation. To recapitulate, we define the derivation of variant-specific delta sets independently from feature configurations F_v based on a delta dependency graph DG_{DM} and a variant tree VT. A delta dependency graph DG_{DM} captures all deltas δ of a delta model DM as delta nodes N_{DM} and their inter-dependencies as dependency edges Dep_{DM} connecting delta nodes. There exist two types of delta dependencies $\mathcal{L}_{\Delta_{Dep}} = \mathcal{L}^{single}_{\Delta_{Dep}} \cup \mathcal{L}^{multi}_{\Delta_{Dep}}$, namely single-target dependencies $\mathcal{L}^{single}_{\Delta_{Dep}}$, e.g., the mandatory dependency *man*, and multi-target dependencies $\mathcal{L}^{multi}_{\Delta_{Dep}}$, e.g., the complete alternative dependency *calt*. The delta dependency graph specifies how deltas of a delta model are related to each other w.r.t. their potential combination for variant-specific delta sets. To derive the set of variant-specific delta sets, we exploit the dependency information captured in the delta dependency graph to create a variant tree VT. A variant tree VT comprises for each delta δ a respective hierarchy level, where

Table 4.4: Symbol Summary of Delta Set Derivation Definition

Symbol	Description
DG_{DM}	Delta dependency graph
$N_{DM}, \mathcal{N}_{\Delta}$	Finite set of delta nodes; Universe of delta nodes
Dep_{DM}	Delta dependency edge relation
$\mathcal{L}_{\Delta_{Dep}}, \mathcal{L}^{single}_{\Delta_{Dep}}, \mathcal{L}^{multi}_{\Delta_{Dep}}$	Finite set of delta dependency labels; Finite set of single-target delta dependency labels; Finite set of multi-target delta dependency labels
man	Mandatory delta dependency
ex	Exclusive delta dependency
pop	Partial optional delta dependency
cop	Complete optional delta dependency
calt	Complete alternative delta dependency
palt	Partial alternative delta dependency
copalt	Complete optional alternative delta dependency
popalt	Partial optional alternative delta dependency
opalt	Optional alternative delta dependency
nexalt	Non-exclusive alternative delta dependency
nexpalt	Non-exclusive partial alternative delta dependency
imp	Implication delta dependency
$VT, \mathcal{VT}$	Variant tree; Universe of variant trees
N_{VT}	Finite set of delta tree nodes
$\prec_L, \prec_R$	Left-child variant tree hierarchy function; Right-child variant tree hierarchy function
α	Variant tree restriction function
λ	Variant tree labeling function
$\prec$	Variant tree hierarchy function
$\rho_{VT}, \Delta_{\rho_{VT}}, \Delta^{\mathcal{VT}}_{\rho_{VT}}$	Variant-tree path; Finite set of variant-tree paths; Universe of variant-tree paths
$\lambda_{\prec}$	Variant-tree path labeling function

respective delta tree nodes $n_\delta \in N_{VT}$ are mapped to the delta $\lambda(n_\delta) = \delta$ and further are connected to their parent delta tree node $n_{\delta'}$ as left $\prec_L (n_{\delta'}) = n_\delta$ or right child $\prec_R (n_{\delta'}) = n_\delta$. If a delta tree node is a left child, the respective delta is selected for a variant-specific delta set. Otherwise, if a delta tree node is a right child, the respective delta is not selected for a variant-specific delta set. Based on the delta dependencies captured in the delta dependency graph, we are able to derive restrictions of delta tree nodes. A restriction indicates that a delta cannot be integrated as left or right child due to dependencies to already integrated deltas. In the end, a variant tree VT specifies the set of derivable delta sets of an SPL version θ_i represented by variant-tree paths $\rho^v_{VT} \in \Delta^{\theta_i}_{\rho_{VT}}$ which starts in a leaf delta tree node and ends in the empty root node.

4.2.3 Incremental Delta Set Derivation

When stepping to the next SPL version θ_{i+1}, we take the changes of a delta model DM_{θ_i} to be transformed into $DM_{\theta_{i+1}}$ which are captured in a higher-order delta $\delta^H_{\theta_{i+1}}$ into account and pass on those changes to variant-specific delta sets Δ_v. Therefore, we exploit the evolution change operation of $\delta^H_{\theta_{i+1}}$ to incrementally adapt the delta dependency graph $DG^{\theta_i}_{DM}$ similar to our slicing approach (cf. Sect. 4.1.2). As result, we obtain the changes to the delta dependency graph, which in turn, are used to incrementally recompute the variant tree $VT_{\theta_{i+1}}$ of SPL version θ_{i+1} based on its previous version VT_{θ_i}. During the recomputation, we capture the differences of the delta sets between both SPL versions facilitating the reasoning about the changes from $\mathbb{V}_{\theta_i}$ to $\mathbb{V}_{\theta_{i+1}}$ on the variant level by identifying removed, added, modified, or unchanged versions of variants. The reasoning process and, hence, the incremental delta dependency graph adaptation as well as incremental variant tree computation are described in the following paragraphs.

Incremental Delta Dependency Graph Adaptation. The application of a higher-order delta $\delta^H_{\theta_{i+1}}$ transforms a delta model DM_{θ_i} into its next version $DM_{\theta_{i+1}}$ by adding, removing, or modifying the encapsulated state machine deltas $\delta \in \Delta_{DM_{\theta_i}}$. We exploit the captured evolution change operations of a higher-order delta to incrementally adapt the delta dependency graph $DG^{\theta_i}_{DM}$ of the previous SPL version θ_i. For the adaptation, we first add and remove delta nodes from the delta dependency graph w.r.t. the additions and removals of state machine deltas. In addition, we remove delta dependency edges which are connected to removed delta nodes. Afterwards, we focus on the delta dependency analysis, where we check if (1) dependency edges are still valid and remain in the graph, (2) are obsolete and have to be removed, or (3) are new and have to be added to the graph. Therefore, we, again, adopt the concept of delta modeling [Sch10; CHS15] and derive respective change operations to transform $DG^{\theta_i}_{DM}$ into $DG^{\theta_{i+1}}_{DM}$. A *delta dependency graph change operation* specifies (1) the addition and removal of delta nodes, and (2) the addition and removal of delta dependency edges.

Definition 4.12: Delta Dependency Graph Change Operation

Let $\mathcal{OP}^{DG_{DM}}$ be the universe of all delta dependency graph change operations defined over the universe $\mathcal{N}_\Delta$ of all delta nodes and the delta dependency edge relation $\mathcal{N}_\Delta \times \mathcal{L}_{\Delta_{Dep}} \times \mathcal{N}^+_\Delta$. The universe $\mathcal{N}_\Delta$ is further defined by the universe of all delta models $\mathcal{DM}$. A *delta dependency graph change operation* $op^{DG_{DM}} \in \mathcal{OP}^{DG_{DM}}$ defines one of the following transformations:

- add n_δ, i.e., a delta node $n_\delta \in \mathcal{N}_\Delta$ is added,
- rem n_δ, i.e., a delta node $n_\delta \in \mathcal{N}_\Delta$ is removed,

- add $(n_\delta, l^{single}_{\Delta_{Dep}}, n_{\delta'})$, i.e., a single-target delta dependency edge $(n_\delta, l^{single}_{\Delta_{Dep}}, n_{\delta'}) \in \mathcal{N}_\Delta \times \mathcal{L}^{single}_{\Delta_{Dep}} \times \mathcal{N}_\Delta$ is added,
- rem $(n_\delta, l^{single}_{\Delta_{Dep}}, n_{\delta'})$, i.e., a single-target delta dependency edge $(n_\delta, l^{single}_{\Delta_{Dep}}, n_{\delta'}) \in \mathcal{N}_\Delta \times \mathcal{L}^{single}_{\Delta_{Dep}} \times \mathcal{N}_\Delta$ is removed,
- add $(n_\delta, l^{multi}_{\Delta_{Dep}}, \{n_{\delta'}, n_{\delta''}\})$, i.e., a multi-target delta dependency edge $(n_\delta, l^{multi}_{\Delta_{Dep}}, \{n_{\delta'}, n_{\delta''}\}) \in \mathcal{N}_\Delta \times \mathcal{L}^{multi}_{\Delta_{Dep}} \times \mathcal{N}^+_\Delta$ is added, and
- rem $(n_\delta, l^{multi}_{\Delta_{Dep}}, \{n_{\delta'}, n_{\delta''}\})$, i.e., a multi-target delta dependency edge $(n_\delta, l^{multi}_{\Delta_{Dep}}, \{n_{\delta'}, n_{\delta''}\}) \in \mathcal{N}_\Delta \times \mathcal{L}^{multi}_{\Delta_{Dep}} \times \mathcal{N}^+_\Delta$ is removed.

We capture the derived change operations in a delta dependency graph regression delta which is afterwards used in the step of incremental variant tree computation.

Definition 4.13: Delta Dependency Graph Regression Delta

Let $\text{apply}_{DG_{DM}}$ be the incremental application function to transform a delta dependency graph $DG^{\theta_i}_{DM}$ of SPL version θ_i into the subsequent delta dependency graph $DG^{\theta_{i+1}}_{DM}$ of SPL version θ_{i+1} based on a given set of delta dependency graph change operations. A *delta dependency graph regression delta* $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}} = \{op^{DG_{DM}}_1, \ldots, op^{DG_{DM}}_m\}$ captures all delta dependency graph change operations such that $DG^{\theta_{i+1}}_{DM} = \text{apply}_{DG_{DM}}(DG^{\theta_i}_{DM}, \Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}})$ holds.

The incremental application function $\text{apply}_{DG_{DM}}$ is defined similar to the functions apply_δ for the incremental state machine delta application (cf. Def. 3.12) as well as apply_{δ^H} for the higher-order delta application (cf. Def. 3.17). So far, we solely incorporate the addition and removal of state machine deltas for the adaptation of the delta dependency graph as modifications of state machine deltas have no influence on the adaptation. We will take the modifications into account during the incremental variant tree computation and reasoning process. Based on those definitions, we propose the incremental delta dependency graph adaptation as follows, where the respective algorithm is shown in Alg. 4.19 in pseudo code. For the adaptation, we use the delta dependency graph $DG^{\theta_i}_{DM}$ of the previous SPL version θ_i, the higher-order delta $\delta^H_{\theta_{i+1}}$, the delta set $\Delta_{DM_{\theta_{i+1}}}$ of the delta model $DM_{\theta_{i+1}}$, and the feature model $fm_{\theta_{i+1}}$ of the new SPL version θ_{i+1} as input. As result, we return the delta dependency graph $DG^{\theta_{i+1}}_{DM}$ of SPL version θ_{i+1} as well as the delta dependency graph regression delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$ capturing the differences between $DG^{\theta_i}_{DM}$ and $DG^{\theta_{i+1}}_{DM}$. The pseudo code algorithms for the auxiliary functions `initDeltaDGRegDelta`, `checkSingleDeltaDeps`, and `checkMultiDeltaDeps` are shown in Alg. 4.20, Alg. 4.21, and Alg. 4.22, respectively.

We start the incremental adaptation by initializing the delta dependency graph regression delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$ (cf. Line. 2). As depicted in Alg. 4.20, we iterate over the evolution change operations captured in the higher-order delta $\delta^H_{\theta_{i+1}}$ and focus on the addition and removal of deltas. For the addition of a delta, we derive a respective addition of its delta node and add the new delta dependency graph change operation to the delta dependency graph regression delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$ to be initialized. Similarly, for the removal of a delta, we derive a respective removal of its delta node and add the change operation to $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$. In addition, we determine for the delta node to be removed, all delta dependencies

Algorithm 4.19.: Incremental Delta Dependency Graph Adaptation

Input: Delta Dependency Graph $DG_{DM}^{\theta_i}$, Higher-Order Delta $\delta_{\theta_{i+1}}^{H}$, Delta Set $\Delta_{DM_{\theta_{i+1}}}$, and Feature Model $fm_{\theta_{i+1}}$

Output: Delta Dependency Graph $DG_{DM}^{\theta_{i+1}}$ and Delta Dependency Graph Regression Delta $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$

Function *incDeltaDepGraphAdapt*
- $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} := initDeltaDGRegDelta(DG_{DM}^{\theta_i}, \delta_{\theta_{i+1}}^{H})$;
- $DG_{DM}^{\theta_{i+1}} := \mathsf{apply}_{DG_{DM}}(DG_{DM}^{\theta_i}, \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}})$;
- $\Delta_{DG_{DM}}^{single} := checkSingleDeltaDeps(DG_{DM}^{\theta_{i+1}}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
- $DG_{DM}^{\theta_{i+1}} := \mathsf{apply}_{DG_{DM}}(DG_{DM}^{\theta_{i+1}}, \Delta_{DG_{DM}}^{single})$;
- $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} := \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} \cup \Delta_{DG_{DM}}^{single}$;
- $\Delta_{DG_{DM}}^{multi} := checkMultiDeltaDeps(DG_{DM}^{\theta_{i+1}}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
- $DG_{DM}^{\theta_{i+1}} := \mathsf{apply}_{DG_{DM}}(DG_{DM}^{\theta_{i+1}}, \Delta_{DG_{DM}}^{multi})$;
- $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} := \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} \cup \Delta_{DG_{DM}}^{multi}$;

return $DG_{DM}^{\theta_{i+1}}, \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$;

Algorithm 4.20.: Function `initDeltaDGRegDelta`

Input: Previous Delta Dependency Graph $DG_{DM}^{\theta_i}$ and Higher-Order Delta $\delta_{\theta_{i+1}}^{H}$

Output: Delta Dependency Graph Regression Delta $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$

Function *initDeltaDGRegDelta*
- $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} := \emptyset$;
- **forall** $op^H \in OP_{\delta_{\theta_{i+1}}^{H}}^{H}$ **do**
  - **if** $op^H == \mathsf{add}\,\delta$ **then**
    - $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} := \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} \cup \{\mathsf{add}\,n_\delta\}$;
  - **else if** $op^H == \mathsf{rem}\,\delta$ **then**
    - $n_\delta := getNode(DG_{DM}^{\theta_i}, \delta)$;
    - $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} := \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} \cup \{\mathsf{rem}\,n_\delta\}$;
    - **forall** $dep \in Dep_{DM}^{DG_{DM}^{\theta_i}}$ **do**
      - **if** $dep == (n_\delta, l_{\Delta_{Dep}}^{single}, n_{\delta'}) \vee dep == (n_{\delta'}, l_{\Delta_{Dep}}^{single}, n_\delta) \vee dep == (n_\delta, l_{\Delta_{Dep}}^{multi}, \{n_{\delta'}, \ldots\}) \vee dep == (n_{\delta'}, l_{\Delta_{Dep}}^{multi}, \{n_\delta, \ldots\})$ **then**
        - $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} := \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}} \cup \{\mathsf{rem}\,dep\}$;

return $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$;

captured in the delta dependency graph $DG_{DM}^{\theta_i}$ to be adapted of the previous SPL version θ_i it is involved in and also derive as well as add remove change operations for those dependencies. Based on the preliminary regression delta $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$, we initially adapt the delta dependency graph $DG_{DM}^{\theta_{i+1}}$ of the new SPL version θ_{i+1} based on its previous version $DG_{DM}^{\theta_i}$ as shown in Line 3 of Alg. 4.20.

Afterwards, we apply a recheck for single- and multi-target delta dependencies for all nodes $n_\delta \in N_{DM}$ of $DG_{DM}^{\theta_{i+1}}$ and capture the differences as delta dependency graph change operations (cf. Line 4ff). In contrast to the incremental dependency graph adaptation of our slicing technique (cf. Sect. 4.1.2), where we have to recheck the dependencies solely for new and change-affected element nodes, we (re-)check for delta dependencies for all delta nodes of a delta dependency graph. This has two reasons. First, each delta $\delta \in \Delta_{DM_{\theta_{i+1}}}$ has relations to all other deltas $\delta' \in \Delta_{DM_{\theta_{i+1}}}$ of a delta model w.r.t. their potential combination for variant-specific delta sets captured as single-target dependencies. Hence, newly added deltas will definitely trigger the recheck for all existing deltas. Second, the evolution from SPL version θ_i to version θ_{i+1} potentially requires only changes in the problem space such that the feature model is evolved [BPD+10; SSA13a; BKL+16; NSS16], but the respective delta models DM_{θ_i} and $DM_{\theta_{i+1}}$ are equal represented by an empty higher-order delta $\delta_{\theta_{i+1}}^H$. In such a case, we have to recheck for potentially changing delta dependencies as due to the feature model evolution [BPD+10; SSA13a; BKL+16; NSS16] the combination of deltas for a variant-specific delta set and, thus, the delta dependencies may change. We record the changes of single- and multi-target dependencies and adapt the delta dependency graph $DG_{DM}^{\theta_{i+1}}$ as well as the delta dependency graph regression delta $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$, accordingly.

For the determination of single-target delta dependencies, as shown in Alg. 4.21, we iterate over all delta nodes $n_\delta \in N_{DM}^{DG_{DM}^{\theta_{i+1}}}$ and check for every other delta node $n_{\delta'} \in N_\Delta$ whether a dependency exist or not. The set N_Δ is used to ensure that every pair of delta nodes is solely evaluated once. If a dependency between both delta nodes n_δ and $n_{\delta'}$ exist, i.e., $dep \neq \dashv$, where $\dashv$ denotes the non-existence, we have to check whether the dependency dep is still valid by taking the respective deltas δ and δ' as well as the feature model $fm_{\theta_{i+1}}$ of the current SPL version θ_{i+1} into account. For the validation check, we refer to the definition of single-target delta dependencies described in Sect. 4.2.2, where three satisfiability checks are made to reason about the dependency type, namely (1) if both deltas are applicable together, (2) δ is applicable without δ', and (3) δ' is applicable without δ. In case the dependency is not valid anymore, e.g., due to the evolution of the feature model, we derive a remove change operation for the obsolete dependency and add it to the set of delta dependency graph change operations $\Delta_{DG_{DM}}^{single}$. Furthermore, we compute the new dependency between both delta nodes similar as for the validation check by performing the three satisfiability checks and the reasoning about the respective result. For the new dependency, we derive an add change operation which is also added to $\Delta_{DG_{DM}}^{single}$. In contrast, if no dependency exist between both delta nodes n_δ and $n_{\delta'}$, i.e., $dep == \dashv$, we directly compute the respective dependency $dep' = (n_\delta, l_{\Delta_{Dep}}^{single}, n_{\delta'})$ and add it to the set of change operations $\Delta_{DG_{DM}}^{single}$.

For the determination of multi-target delta dependencies, as shown in Alg. 4.22, we first iterate over all existing multi-target dependencies $dep \in Dep_{DM}^{multi}$ to identify obsolete once. Therefore, we check for each dependency dep whether it is still valid by incorporating the set of single-target delta dependencies as well as the set of deltas $\Delta_{DM_{\theta_{i+1}}}$ and the feature model $fm_{\theta_{i+1}}$. For the validation check, we refer to the definition of multi-target delta dependencies described in Sect. 4.2.2. A

Algorithm 4.21.: Function `checkSingleDeltaDeps`

Input: Intermediate Delta Dependency Graph $DG_{DM}^{\theta_{i+1}}$, Set of Deltas $\Delta_{DM_{\theta_{i+1}}}$, Feature Model $fm_{\theta_{i+1}}$

Output: Set of Delta Dependency Graph Change Operations $\Delta_{DG_{DM}}^{single}$

```
1  Function checkSingleDeltaDeps
2      Δ^single_DG_DM := ∅;
3      N_Δ = N_DM^(DG_DM^θ_{i+1}) ;
4      forall n_δ ∈ N_DM^(DG_DM^θ_{i+1}) do
5          N_Δ := N_Δ \ {n_δ};
6          forall n_δ' ∈ N_Δ do
7              dep := getSingleDependency(DG_DM^θ_{i+1}, n_δ, n_δ');
8              δ := getDelta(Δ_DM_θ_{i+1}, n_δ);
9              δ' := getDelta(Δ_DM_θ_{i+1}, n_δ');
10             if dep ≠ ⊣ then
11                 if !valid(dep, δ, δ', fm_θ_{i+1}) then
12                     Δ^single_DG_DM := Δ^single_DG_DM ∪ {rem dep};
13                     dep' := computeDependency(n_δ, n_δ', δ, δ', fm_θ_{i+1});
14                     Δ^single_DG_DM := Δ^single_DG_DM ∪ {add dep'};
15             else
16                 dep' := computeDependency(n_δ, n_δ', δ, δ', fm_θ_{i+1});
17                 Δ^single_DG_DM := Δ^single_DG_DM ∪ {add dep'};
18 return Δ^single_DG_DM;
```

multi-target dependency becomes invalid and, hence, obsolete if its prerequisites are not fulfilled anymore, i.e., the set of single-target delta dependencies and/or the feature model do not allow for the combination of the respective deltas, the multi delta dependency describes. In case we detect an invalid dependency, we derive a respective remove change operation and add it to the set of delta dependency change operations $\Delta_{DG_{DM}}^{multi}$. Afterwards, we have to determine the set of new multi-target delta dependencies by iterating over all delta nodes $n_\delta \in N_{DM}^{DG_{DM}^{\theta_{i+1}}}$ of the delta dependency graph. For every delta node n_δ, we determine the set of single-target delta dependencies it is involved in and capture them in the set $Dep_{DM}^{n_\delta}$. Based on this set, we are able to compute the different multi-target delta dependencies if existing, where the current delta node defines the source node of the dependency. We refer to the definition of the multi-target dependencies in Sect. 4.2.2 for their computation. As the last step, we iterate over all computed dependencies and check if they are already part of the delta dependency graph. If not, we derive an add change operation and add it to $\Delta_{DG_{DM}}^{multi}$.

The incremental adaptation of the delta dependency graph terminates since the set N_{DM} of delta nodes captured in the dependency graph $DG_{DM}^{\theta_{i+1}}$ is finite. As result of the adaptation, we obtain the valid dependency graph $DG_{DM}^{\theta_{i+1}}$ of the current SPL version θ_{i+1} under analysis as well as the differences to the delta dependency graph $DG_{DM}^{\theta_i}$ of the previously analyzed SPL version θ_i recorded in a delta dependency graph regression delta $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$. We use both as input for the incremental computation of the variant tree as described in the following.

Algorithm 4.22.: Function `checkMultiDeltaDeps`

Input: Intermediate Delta Dependency Graph $DG_{DM}^{\theta_{i+1}}$, Set of Deltas $\Delta_{DM_{\theta_{i+1}}}$, Feature Model $fm_{\theta_{i+1}}$
Output: Set of Delta Dependency Graph Change Operations $\Delta_{DG_{DM}}^{multi}$

Function *checkMultiDeltaDeps*
  $\Delta_{DG_{DM}}^{multi} := \emptyset$;
  $Dep_{DM}^{single} := getSingleDependencies(DG_{DM}^{\theta_{i+1}})$;
  $Dep_{DM}^{multi} := getMultiDependencies(DG_{DM}^{\theta_{i+1}})$;
  forall $dep \in Dep_{DM}^{multi}$ **do**
    if $!valid(dep, Dep_{DM}^{single}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$ **then**
      $\Delta_{DG_{DM}}^{multi} := \Delta_{DG_{DM}}^{multi} \cup \{\mathsf{rem}\ dep\}$;
  forall $n_\delta \in N_{DM}^{DG_{DM}^{\theta_{i+1}}}$ **do**
    $Dep_{DM}^{n_\delta} := getDependencies(Dep_{DM}^{single}, n_\delta)$;
    $Dep_{DM}^{calt} := computeCompleteAlternativeDep(n_\delta, Dep_{DM}^{n_\delta}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
    $Dep_{DM}^{palt} := computePartialAlternativeDep(n_\delta, Dep_{DM}^{n_\delta}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
    $Dep_{DM}^{popalt} := computePartialOptionalAlternativeDep(n_\delta, Dep_{DM}^{n_\delta}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
    $Dep_{DM}^{copalt} := computeCompleteOptionalAlternativeDep(n_\delta, Dep_{DM}^{n_\delta}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
    $Dep_{DM}^{opalt} := computeOptionalAlternativeDep(n_\delta, Dep_{DM}^{n_\delta}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
    $Dep_{DM}^{nexalt} := computeNonExclusiveAlternativeDep(n_\delta, Dep_{DM}^{n_\delta}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
    $Dep_{DM}^{nexpalt} := computeNonExclusivePartialAlternativeDep(n_\delta, Dep_{DM}^{n_\delta}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
    $Dep_{DM}^{imp} := computeImplicationDep(n_\delta, Dep_{DM}^{n_\delta}, \Delta_{DM_{\theta_{i+1}}}, fm_{\theta_{i+1}})$;
    $Dep_{DM} := Dep_{DM}^{calt} \cup Dep_{DM}^{palt} \cup Dep_{DM}^{popalt} \cup Dep_{DM}^{copalt} \cup Dep_{DM}^{opalt} \cup Dep_{DM}^{nexalt} \cup Dep_{DM}^{nexpalt} \cup Dep_{DM}^{imp}$;
    forall $dep \in Dep_{DM}$ **do**
      if $dep \notin Dep_{DM}^{multi}$ **then**
        $\Delta_{DG_{DM}}^{multi} := \Delta_{DG_{DM}}^{multi} \cup \{\mathsf{add}\ dep\}$;
return $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$;

Example 4.7: Incremental Delta Dependency Graph Adaptation

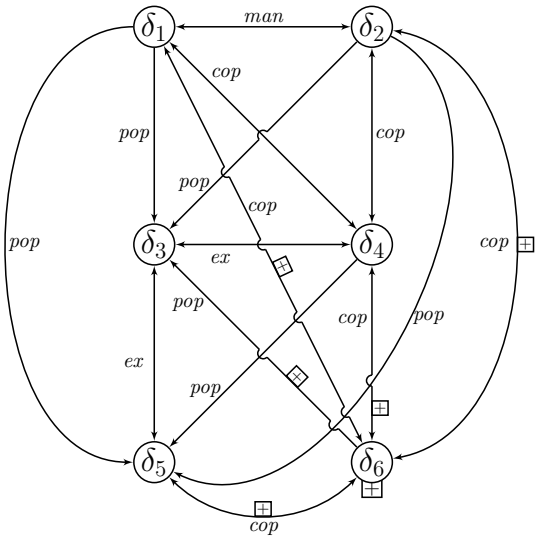
Consider the delta dependency graph $DG_{DM}^{\theta_1}$ of delta model DM_{θ_1} from Ex. 3.5 for SPL version θ_1 of our running example shown in Fig. 4.10a w.r.t. single-target dependencies and in Fig. 4.10b w.r.t. multi-target dependencies. Based on the application of $\delta_{\theta_1}^H$ from Ex. 3.5, the delta set and, thus, the dependencies between deltas change. Compared to the previous graph $DG_{DM}^{\theta_0}$, the incremental delta dependency graph adaptation adds the delta node n_{δ_6} for delta δ_6, the single-target dependencies $(\delta_6, cop, \delta_1)$, $(\delta_6, cop, \delta_2)$, $(\delta_6, pop, \delta_3)$, $(\delta_6, cop, \delta_4)$, and $(\delta_6, cop, \delta_5)$ as well as the multi-target dependencies $(\delta_6, popalt, \{\delta_3, \delta_5\})$, $(\delta_6, popalt, \{\delta_3, \delta_4\})$, and $(\delta_4, nexpalt, \{\delta_5, \delta_6\})$. The evolution step from SPL version θ_0 to version θ_1 requires no removal of delta nodes or delta dependency edges. In both figures, the differences of the delta dependency graph $DG_{DM}^{\theta_1}$ to its previous version $DG_{DM}^{\theta_0}$ are highlighted by a $+$. The resulting delta dependency graph regression delta is defined as $\Delta_{\theta_0,\theta_1}^{DG_{DM}} = \{\mathsf{add}\ n_{\delta_6}, \mathsf{add}\ (\delta_6, cop, \delta_1), \mathsf{add}$

$(\delta_6, cop, \delta_2)$, add $(\delta_6, pop, \delta_3)$, add $(\delta_6, cop, \delta_4)$, add $(\delta_6, cop, \delta_5)$, add $(\delta_6, popalt, \{\delta_3, \delta_5\})$, add $(\delta_6, popalt, \{\delta_3, \delta_4\})$, add $(\delta_4, nexpalt, \{\delta_5, \delta_6\})\}$.

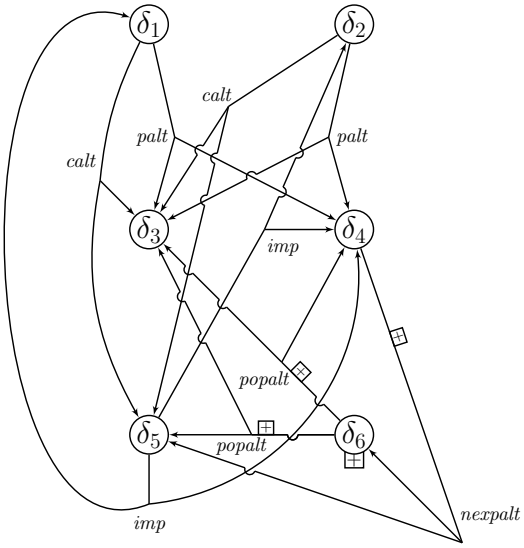
When stepping to SPL version θ_2, there are no changes to the delta model DM_{θ_1} to obtain DM_{θ_2} as the higher-order delta $\delta^H_{\theta_2}$ is empty (cf. Ex. 3.5). The respective delta dependency graph has still to be adapted by rechecking the dependencies between deltas $\delta \in \Delta_{DM_{\theta_2}}$. During the incremental adaptation, we remove the single-target dependency $(\delta_6, pop, \delta_3)$ and add $(\delta_6, cop, \delta_3)$. In addition, we remove the multi-target dependencies $(\delta_6, popalt, \{\delta_3, \delta_5\})$ as well as $(\delta_6, popalt, \{\delta_3, \delta_4\})$ and add the dependencies $(\delta_6, opalt, \{\delta_3, \delta_5\})$ and $(\delta_6, opalt, \{\delta_3, \delta_4\})$. The delta dependency graph $DG^{\theta_2}_{DM}$ is depicted in Fig. 4.10c with its contained single-target delta dependencies and in Fig. 4.10d with its multi-target delta dependencies. In both figures, the removal and addition of the single- as well as multi-target dependencies are combined and highlighted by annotating the graph elements with a $*$. The resulting delta dependency graph regression delta is defined as $\Delta^{DG_{DM}}_{\theta_1,\theta_2} = \{$rem $(\delta_6, pop, \delta_3)$, add $(\delta_6, cop, \delta_3)$, rem $(\delta_6, popalt, \{\delta_3, \delta_5\})$, rem $(\delta_6, popalt, \{\delta_3, \delta_4\})$, add $(\delta_6, opalt, \{\delta_3, \delta_5\})$, add $(\delta_6, opalt, \{\delta_3, \delta_4\})\}$.

Incremental Variant Tree Generation. To reason about the higher-order delta application and, hence, how the variant set $\mathbb{V}_{\theta_i}$ of the previous SPL version θ_i changes to $\mathbb{V}_{\theta_{i+1}}$ of the new SPL version θ_{i+1} w.r.t. the derivable delta sets Δ_v, we incrementally compute the variant tree $VT_{\theta_{i+1}}$ based on its previous version VT_{θ_i}. At the same time, we determine whether a variant v is modified, added, removed, or remains unchanged by investigating the impact on its respective delta set Δ_v. We propose the incremental variant tree computation as follows, where the corresponding algorithm is shown in Alg. 4.23 in pseudo code. For the computation, we require the previous variant tree version VT_{θ_i} as well as its derivable set $\Delta^{\theta_i}_{\rho_{VT}}$ of variant-tree paths, the adapted delta dependency graph $DG^{\theta_{i+1}}_{DM}$, the delta dependency graph regression delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$, the higher-order delta $\delta^H_{\theta_{i+1}}$, and the current delta set $\Delta_{DM_{\theta_{i+1}}}$ of the delta model $DM_{\theta_{i+1}}$ as input. As result, we provide the valid variant tree $VT_{\theta_{i+1}}$ for the current SPL version θ_{i+1} under consideration, the categorized sets of new $\Delta^{new}_{\rho_{VT}}$, modified $\Delta^{mod}_{\rho_{VT}}$, and unchanged $\Delta^{\emptyset}_{\rho_{VT}}$ variant-tree paths, and also a mapping $\triangleleft(\rho_{VT}) = \rho'_{VT}$ between the tree paths $\rho_{VT} \in \Delta^{\theta_i}_{\rho_{VT}}$ of the previous SPL version θ_i and those paths $\rho'_{VT} \in \Delta^{\theta_{i+1}}_{\rho_{VT}}$ of the subsequent SPL version θ_{i+1}. Therefore, we define the mapping function $\triangleleft : \Delta^{\mathcal{VT}}_{\rho_{VT}} \rightarrow \Delta^{\mathcal{VT}}_{\rho_{VT}} \cup \emptyset$ such that each tree path of the previous SPL version has a respective successor in the subsequent SPL version or is mapped to the empty set indicating that the variant-tree path and, hence, its represented variant is removed. By $\Delta^{\mathcal{VT}}_{\rho_{VT}}$, we refer to the universe of all variant-tree paths which is defined over the universe $\mathcal{VT}$ of variant trees. The categorized sets as well as the path mapping indicate the change impact of the higher-order delta application and are used to derive a variant set evolution delta $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$ which, in turn, is exploited in our model-based regression testing framework to guide the retest test selection after an SPL evolves to its next version (cf. Chapt. 5). The pseudo code algorithms of the auxiliary functions `remObsoleteRestrictions`, `completeTree`, `remObsoleteNodes`, `updatePathMappingAfterRemoval`, `determineNewRestrictions`, `integrateNewDeltas`, `updatePathMappingAfterAddition`, and `incorporateModDeltas` are shown in Alg. 4.24, Alg. 4.25, Alg. 4.26, Alg. 4.27, Alg. 4.28, Alg. 4.29, Alg. 4.30, and Alg. 4.31, respectively.

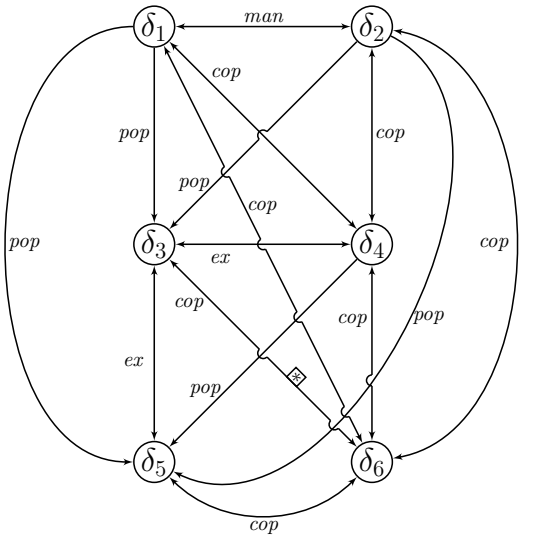
We start the incremental tree computation (cf. Line 3) by categorizing all existing tree paths $\rho^v_{VT} \in \Delta^{\theta_i}_{\rho_{VT}}$ as unchanged $\Delta^{\emptyset}_{\rho_{VT}}$ and initialize the mapping $\triangleleft$ accordingly such that $\forall \rho^v_{VT} \in \Delta^{\emptyset}_{\rho_{VT}} : \triangleleft(\rho^v_{VT}) =$



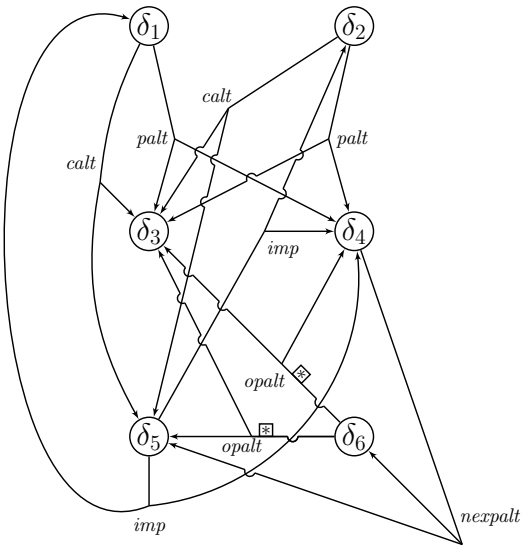
(a) Delta Dependency $DG_{DM}^{\theta_1}$ Graph with Single-Target Dependencies



(b) Delta Dependency $DG_{DM}^{\theta_1}$ Graph with Multi-Target Dependencies



(c) Delta Dependency $DG_{DM}^{\theta_2}$ Graph with Single-Target Dependencies



(d) Delta Dependency $DG_{DM}^{\theta_2}$ Graph with Multi-Target Dependencies

Figure 4.10: Delta Dependency Graphs $DG_{DM}^{\theta_1}$ and $DG_{DM}^{\theta_2}$ for SPL Versions θ_1 and θ_2 Including Differences to the Previous Delta Dependency Graphs $DG_{DM}^{\theta_0}$ and $DG_{DM}^{\theta_1}$

Algorithm 4.23.: Incremental Variant Tree Computation

Input: Variant Tree VT_{θ_i}, Set of Variant-Tree Paths $\Delta^{\theta_i}_{\rho_{VT}}$, Delta Dependency Graph $DG^{\theta_{i+1}}_{DM}$, Delta Dependency Graph Regression Delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$, Higher-Order Delta $\delta^{H}_{\theta_{i+1}}$, and Delta Set $\Delta_{DM_{\theta_{i+1}}}$

Output: Variant Tree $VT_{\theta_{i+1}}$, Set of New Variant-Tree Paths $\Delta^{new}_{\rho_{VT}}$, Set of Modified Variant-Tree Paths $\Delta^{mod}_{\rho_{VT}}$, Set of Unchanged Variant-Tree Paths $\Delta^{\emptyset}_{\rho_{VT}}$, and Variant-Tree Path Mapping $\triangleleft$

1 **Function** *incBuildVariantTree*
2 $VT_{\theta_{i+1}} := VT_{\theta_i}$;
3 $\Delta^{\emptyset}_{\rho_{VT}} := initializePathMapping(\Delta^{\theta_i}_{\rho_{VT}}, \triangleleft)$;
4 $\Delta^{new}_{\rho_{VT}} := \emptyset$;
5 $\Delta^{mod}_{\rho_{VT}} := \emptyset$;
6 $N^{unrestrict}_{VT} := remObsoleteRestrictions(VT_{\theta_{i+1}}, \Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}})$;
7 $VT_{\theta_{i+1}} := completeTree(VT_{\theta_{i+1}}, N^{unrestrict}_{VT}, DG^{\theta_{i+1}}_{DM}, \Delta_{DM_{\theta_{i+1}}}, \Delta^{\emptyset}_{\rho_{VT}}, \Delta^{new}_{\rho_{VT}})$;
8 $VT_{\theta_{i+1}} := remObsoleteNodes(VT_{\theta_{i+1}}, \delta^{H}_{\theta_{i+1}}, \Delta^{new}_{\rho_{VT}}, \Delta^{mod}_{\rho_{VT}}, \Delta^{\emptyset}_{\rho_{VT}}, \triangleleft)$;
9 $VT_{\theta_{i+1}} := determineNewRestrictions(VT_{\theta_{i+1}}, DG^{\theta_{i+1}}_{DM}, \Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}, \Delta_{DM_{\theta_{i+1}}}, \Delta^{new}_{\rho_{VT}}, \Delta^{mod}_{\rho_{VT}}, \Delta^{\emptyset}_{\rho_{VT}}, \triangleleft)$;
10 $VT_{\theta_{i+1}} := integrateNewDeltas(VT_{\theta_{i+1}}, DG^{\theta_{i+1}}_{DM}, \delta^{H}_{\theta_{i+1}}, \Delta_{DM_{\theta_{i+1}}}, \Delta^{new}_{\rho_{VT}}, \Delta^{mod}_{\rho_{VT}}, \Delta^{\emptyset}_{\rho_{VT}}, \triangleleft)$;
11 $incorporateModDeltas(VT_{\theta_{i+1}}, \delta^{H}_{\theta_{i+1}}, \Delta^{mod}_{\rho_{VT}}, \Delta^{\emptyset}_{\rho_{VT}}, \triangleleft)$;
12 **return** $VT_{\theta_{i+1}}, \Delta^{new}_{\rho_{VT}}, \Delta^{mod}_{\rho_{VT}}, \Delta^{\emptyset}_{\rho_{VT}}$;

ρ^{v}_{VT} holds. Of course, the initial categorization and mapping are not fixed and may change during the remaining variant tree computation.

Algorithm 4.24.: Function `remObsoleteRestrictions`

Input: Variant Tree VT_{θ_i} and Delta Dependency Graph Regression Delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$

Output: Set of Unrestricted Delta Tree Nodes $N^{unrestrict}_{VT}$

1 **Function** *remObsoleteRestrictions*
2 $Dep^{rem}_{DM} := \emptyset$;
3 $N^{unrestrict}_{VT} := \emptyset$;
4 **forall** $op^{DG_{DM}} \in \Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$ **do**
5 **if** $op^{DG_{DM}} ==$ rem dep **then**
6 $Dep^{rem}_{DM} := Dep^{rem}_{DM} \cup \{dep\}$;
7 **forall** $n^{\otimes}_{\delta} \in N^{\otimes}_{\Delta}$ **do**
8 $\alpha(n^{\otimes}_{\delta}) := \alpha(n^{\otimes}_{\delta}) \setminus Dep^{rem}_{DM}$;
9 **if** $\alpha(n^{\otimes}_{\delta}) == \emptyset$ **then**
10 $N^{unrestrict}_{VT} := N^{unrestrict}_{VT} \cup \{n^{\otimes}_{\delta}\}$;
11 **return** $N^{unrestrict}_{VT}$;

Removal of Obsolete Restrictions. Afterwards, we remove all obsolete restrictions w.r.t. remove operations of the respective delta dependencies captured in the delta dependency regression delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$ (cf. Line 6). Restrictions between deltas become obsolete (1) due to modifications of their application conditions or changes to the feature model such that they are combinable in a different

way for variant-specific delta sets as in the previous SPL version, or (2) due to the removal of a delta and all its connecting dependency edges which are defining the obsolete restrictions. For the removal of the obsolete restrictions, as shown in Alg. 4.24, we first determine all remove operations for delta dependencies and record the removed dependencies in the set Dep_{DM}^{rem}. In a variant tree, restrictions are used to constrain the creation of invalid paths representing invalid variant-specific delta sets. Hence, we iterate over the set of restricted delta tree nodes $n_\delta^\otimes \in N_\Delta^\otimes$ and remove all obsolete dependencies Dep_{DM}^{rem} from their list of restrictions $\alpha(n_\delta^\otimes)$. After the removal, there are potentially some restricted delta tree nodes $n_\delta^\otimes$ which are not restricted anymore as their list of restricting delta dependencies $\alpha(n_\delta^\otimes) = \emptyset$ is empty. We record those delta tree nodes in the set $N_{VT}^{unrestrict}$ as we are now able to compute respective subtrees for them.

Variant Tree Completion. The subtree completion is performed similar to the standard variant tree computation described above (cf. Alg. 4.15, Line 4 to 21). Hence, as shown in Alg. 4.25, we iterate over all delta tree nodes $n_\delta^\otimes \in N_{VT}^{unrestrict}$, where for each node $n_\delta^\otimes$, we update the relation to its parent node n_δ^{parent} by removing it and adding a new delta tree node n_δ for the delta δ either as left or right child. As next step, we determine the tree path starting in the new node n_δ to identify the deltas which are already integrated in the higher hierarchy levels of the variant tree and the set of deltas which have to be potentially added under n_δ to compute its subtree, where we incorporate the existing set of delta dependencies Dep_{DM} of the current delta dependency graph $DG_{DM}^{\theta_{i+1}}$ to determine potential restrictions. In case the computation of a subtree is required, we perform the same steps as for the standard variant tree computation (cf. Line 18 to 32) and continue with the next delta tree node from the set $N_{VT}^{unrestrict}$ of unrestricted nodes until all nodes are handled. As last step of the tree completion, we derive the set of variant tree paths for the adapted variant tree and capture all paths which are not categorized as unchanged as new tree paths. New paths are now derivable due to the removal of obsolete restrictions. As those paths are completely new, they do not have corresponding predecessor tree paths in $\Delta_{\rho_{VT}}^{\theta_i}$ indicating that an update of the mapping $\triangleleft$ is not required.

Removal of Obsolete Delta Tree Nodes. Afterwards, we remove obsolete delta tree nodes $n_\delta \in N_\Delta \cup N_\Delta^\otimes$ which are mapped to deltas that are removed via the higher-order delta $\delta_{\theta_{i+1}}^H$ (cf. Line 8). Based on the removal of obsolete restrictions, there are no restrictions left which are dependent on a delta tree node n_δ to be removed. That means, that the left as well as right subtree starting from n_δ which may be created via the subtree completion are independent whether the respective delta δ is selected for a variant-specific delta set or not. Hence, the left as well as right subtree are equal as the only restrictions which are contained in the subtrees are introduced based on delta dependencies between deltas in a lower or higher hierarchy level of the variant tree. Based on this situation, a delta tree node $n_\delta \in N_\Delta \cup N_\Delta^\otimes$ is removed by altering the variant tree such that we rehang either the left or right subtree directly under the parent tree node of n_δ. In this thesis, we select the left subtree. As shown in Alg. 4.26, we first identify all deltas which are removed via the $\delta_{\theta_{i+1}}^H$. For each removed delta, we determine its respective variant tree nodes in $VT_{\theta_{i+1}}$ as those nodes have to be removed from the tree. Therefore, we require the parent node n_δ^{parent} as well as the left child tree node n_δ^{left} and right child tree node n_δ^{right} of each n_δ to be removed in order to rehang the respective subtree such that $\prec_L (n_\delta^{parent}) = n_\delta^{left}$ and $\prec_r (n_\delta^{parent}) = n_\delta^{right}$ holds.

The removal of a delta tree node has an impact on the set of derivable variant-tree paths, their preliminary categorization, and their mapping to previous tree paths. As shown in Alg. 4.27, we iterate over all existing variant tree paths to determine the impact of the node removal and adapt the

Algorithm 4.25.: Function `completeTree`

Input: Variant Tree $VT_{\theta_{i+1}}$, Set of Unrestricted Delta Tree Nodes $N_{VT}^{unrestrict}$, Delta Dependency Graph $DG_{DM}^{\theta_{i+1}}$, Delta Set $\Delta_{DM_{\theta_{i+1}}}$, Set of Unchanged Variant-Tree Paths $\Delta_{\rho_{VT}}^{\emptyset}$, and Set of New Variant-Tree Paths $\Delta_{\rho_{VT}}^{new}$

Output: Updated Variant Tree $VT_{\theta_{i+1}}$

1 **Function** *completeTree*
2 **forall** $n_\delta^\otimes \in N_{VT}^{unrestrict}$ **do**
3 $N_\Delta^\otimes := N_\Delta^\otimes \setminus \{n_\delta\}$;
4 $\delta := getDelta(\Delta_{DM_{\theta_{i+1}}}, n_\delta^\otimes)$;
5 $N_\Delta := N_\Delta \cup \{n_\delta\}$;
6 $n_\delta^{parent} := getParent(VT_{\theta_{i+1}}, n_\delta^\otimes)$;
7 **if** $n_\delta^\otimes == \prec_L (n_\delta^{parent})$ **then**
8 $\prec_L (n_\delta^{parent}) := n_\delta$;
9 **else**
10 $\prec_R (n_\delta^{parent}) := n_\delta$;
11 $\prec_L (n_\delta) := \prec_R (n_\delta) := \bot$;
12 $\alpha(n_\delta) := \emptyset$;
13 $\lambda(n_\delta) := \delta$;
14 $\rho_{VT} := getPath(VT_{\theta_{i+1}}, n_\delta)$;
15 $\Delta := \Delta_{DM_{\theta_{i+1}}} \setminus getDeltasFromPath(\Delta_{DM_{\theta_{i+1}}}, \rho_{VT})$;
16 $level := getLevelForNode(n_\delta)$;
17 **if** $\Delta \neq \emptyset$ **then**
18 **forall** $\delta' \in \Delta$ **do**
19 $N_\Delta^{level} := getLeafs(VT_{\theta_{i+1}}, level)$;
20 **forall** $n_{\delta''} \in N_\Delta^{level}$ **do**
21 $Dep_{DM}^{\delta',L} := \emptyset$;
22 $Dep_{DM}^{\delta',R} := \emptyset$;
23 $\rho_{VT} := getPath(VT_{\theta_{i+1}}, n_{\delta''})$;
24 $Dep_{DM}^{\delta',L} := checkDepForRestriction(VT_{\theta_{i+1}}, DG_{DM}^{\theta_{i+1}}, \rho_{VT}, \delta', \texttt{LEFT})$;
25 $Dep_{DM}^{\delta',R} := checkDepForRestriction(VT_{\theta_{i+1}}, DG_{DM}^{\theta_{i+1}}, \rho_{VT}, \delta', \texttt{RIGHT})$;
26 **if** $Dep_{DM}^{\delta',L} == \emptyset \wedge Dep_{DM}^{\delta',R} == \emptyset$ **then**
27 $VT := addChild(VT_{\theta_{i+1}}, n_{\delta'}, \texttt{LEFT\_RIGHT}, \delta', \emptyset)$;
28 **else if** $Dep_{DM}^{\delta',L} == \emptyset \wedge Dep_{DM}^{\delta',R} \neq \emptyset$ **then**
29 $VT := addChild(VT_{\theta_{i+1}}, n_{\delta'}, \texttt{LEFT}, \delta', Dep_{DM}^{\delta})$;
30 **else if** $Dep_{DM}^{\delta',L} \neq \emptyset \wedge Dep_{DM}^{\delta',R} == \emptyset$ **then**
31 $VT := addChild(VT_{\theta_{i+1}}, n_{\delta'}, \texttt{RIGHT}, \delta', Dep_{DM}^{\delta})$;
32 *level*++;
33 $\Delta_{\rho_{VT}} := derivePathsFromTree(VT_{\theta_{i+1}})$;
34 $\Delta_{\rho_{VT}}^{new} := \Delta_{\rho_{VT}} \setminus \Delta_{\rho_{VT}}^{\emptyset}$;
35 **return** $VT_{\theta_{i+1}}$;

Algorithm 4.26.: Function `remObsoleteNodes`

Input: Variant Tree $VT_{\theta_{i+1}}$, Higher-Order Delta $\delta^H_{\theta_{i+1}}$, Set of New Variant-Tree Paths $\Delta^{new}_{\rho_{VT}}$, Set of Modified Variant-Tree Paths $\Delta^{mod}_{\rho_{VT}}$, Set of Unchanged Variant-Tree Paths $\Delta^{\emptyset}_{\rho_{VT}}$, Variant-Tree Path Mapping $\triangleleft$

Output: Variant Tree $VT_{\theta_{i+1}}$

1 **Function** *remObsoleteNodes*
2 $\quad \Delta_{rem} := \emptyset$;
3 $\quad$ **forall** $op^H \in \delta^H_{\theta_{i+1}}$ **do**
4 $\quad\quad$ **if** $op^H == \mathsf{rem}\,\delta$ **then**
5 $\quad\quad\quad \Delta_{rem} := \Delta_{rem} \cup \{\delta\}$;
6 $\quad$ **forall** $\delta \in \Delta_{rem}$ **do**
7 $\quad\quad N^{\delta}_{VT} := getDeltaTreeNodes(VT_{\theta_{i+1}}, \delta)$;
8 $\quad\quad$ **forall** $n_\delta \in N^{\delta}_{VT}$ **do**
9 $\quad\quad\quad n^{parent}_{\delta} := getParent(VT_{\theta_{i+1}}, n_\delta)$;
10 $\quad\quad\quad n^{left}_{\delta} :=\prec_L (n_\delta)$;
11 $\quad\quad\quad n^{right}_{\delta} :=\prec_R (n_\delta)$;
12 $\quad\quad\quad \prec_L (n^{parent}_{\delta}) := n^{left}_{\delta}$;
13 $\quad\quad\quad \prec_R (n^{parent}_{\delta}) := n^{right}_{\delta}$;
14 $\quad\quad\quad N_{VT_{\theta_{i+1}}} := N_{VT_{\theta_{i+1}}} \setminus \{n_\delta\}$;
15 $\quad\quad\quad updatePathMappingAfterRemoval(n_\delta, \Delta^{new}_{\rho_{VT}}, \Delta^{mod}_{\rho_{VT}}, \Delta^{\emptyset}_{\rho_{VT}}, \triangleleft)$;
16 **return** $VT_{\theta_{i+1}}$;

categorization as well as the mapping accordingly. For unchanged, modified as well as new variant-tree paths ρ^v_{VT} that comprises the removed delta tree node n_δ as right child $\rho^v_{VT} = ((\ldots, n^R_\delta, \ldots), \lambda_\prec)$, the respective sequence is changed by removing n_δ. In this scenario, we update, for unchanged and modified paths, the existing mapping for previous paths such that the function $\triangleleft$ now points to the adapted tree paths. As the incorporation as right child implies that the mapped delta δ is not selected for a variant-specific delta set which is defined by the tree path, the current categorization by means of new, modified, and unchanged tree paths is still valid.

In contrast, for variant-tree paths ρ^v_{VT} that comprises the removed delta tree node n_δ as left child $\rho^v_{VT} = ((\ldots, n^L_\delta, \ldots), \lambda_\prec)$, we have to examine two cases in order to update the categorization as well as the mapping $\triangleleft$. First, the removal of the delta tree node from a tree path results in another tree path already existing in the set $\Delta^{\theta_{i+1}}_{\rho_{VT}}$ of all derivable tree paths, i.e., $\Delta^{\emptyset}_{\rho_{VT}} \cup \Delta^{new}_{\rho_{VT}} \cup \Delta^{mod}_{\rho_{VT}}$, where the delta node was contained as right child before its removal. In this case, the variant which is represented by the old variant-tree path is removed, whereas the existing path is not affected and its categorization is unchanged. We update the mapping for an affected tree path such that the function $\triangleleft$ now points to the empty set. Second, the removal of the tree node results in a tree path which was not derivable until now. Hence, new variant-tree paths $\rho^v_{VT} \in \Delta^{new}_{\rho_{VT}}$ are still categorized as new and modified paths $\rho^v_{VT} \in \Delta^{mod}_{\rho_{VT}}$ are still categorized as modified. In contrast, a previously as unchanged categorized variant-tree path $\rho^v_{VT} \in \Delta^{\emptyset}_{\rho_{VT}}$ is now classified as modified and, therefore, added to the respective set $\Delta^{mod}_{\rho_{VT}}$ of modified tree paths. Again, we update the mapping for an old tree path such that the function $\triangleleft$ now points to the adapted tree path. After the removal of obsolete delta tree

Algorithm 4.27.: Function `updatePathMappingAfterRemoval`

Input: Removed Delta Tree Node n_δ, Set of New Variant-Tree Paths $\Delta^{new}_{\rho_{VT}}$, Set of Modified Variant-ree Paths $\Delta^{mod}_{\rho_{VT}}$, Set of Unchanged Variant-Tree Paths $\Delta^{\emptyset}_{\rho_{VT}}$, and Variant-Tree Path Mapping $\triangleleft$

1 **Function** *updatePathMappingAfterRemoval*
2 $\quad \Delta^{left}_{\rho_{VT}} := \emptyset$;
3 $\quad$ **forall** $\rho_{VT} \in \Delta^{\emptyset}_{\rho_{VT}} \cup \Delta^{new}_{\rho_{VT}} \cup \Delta^{mod}_{\rho_{VT}}$ **do**
4 $\quad\quad$ **if** $n_\delta \in \rho_{VT} \wedge \lambda^{\rho_{VT}}_{\prec}(n_\delta) == R$ **then**
5 $\quad\quad\quad \rho'_{VT} := removeNodeFromSequence(\rho_{VT}, n_\delta)$;
6 $\quad\quad\quad$ **if** $\rho_{VT} \in \Delta^{\emptyset}_{\rho_{VT}}$ **then**
7 $\quad\quad\quad\quad \Delta^{\emptyset}_{\rho_{VT}} := (\Delta^{\emptyset}_{\rho_{VT}} \setminus \{\rho_{VT}\}) \cup \{\rho'_{VT}\}$;
8 $\quad\quad\quad\quad updateMapping(\triangleleft, \rho_{VT}, \rho'_{VT})$
9 $\quad\quad\quad$ **else if** $\rho_{VT} \in \Delta^{new}_{\rho_{VT}}$ **then**
10 $\quad\quad\quad\quad \Delta^{new}_{\rho_{VT}} := (\Delta^{new}_{\rho_{VT}} \setminus \{\rho_{VT}\}) \cup \{\rho'_{VT}\}$;
11 $\quad\quad\quad$ **else**
12 $\quad\quad\quad\quad \Delta^{mod}_{\rho_{VT}} := (\Delta^{mod}_{\rho_{VT}} \setminus \{\rho_{VT}\}) \cup \{\rho'_{VT}\}$;
13 $\quad\quad\quad\quad updateMapping(\triangleleft, \rho_{VT}, \rho'_{VT})$
14 $\quad\quad$ **else if** $n_\delta \in \rho_{VT} \wedge \lambda^{\rho_{VT}}_{\prec}(n_\delta) == L$ **then**
15 $\quad\quad\quad \Delta^{left}_{\rho_{VT}} := \Delta^{left}_{\rho_{VT}} \cup \{\rho_{VT}\}$;
16 $\quad$ **forall** $\rho_{VT} \in \Delta^{left}_{\rho_{VT}}$ **do**
17 $\quad\quad \rho'_{VT} := removeNodeFromSequence(\rho_{VT}, n_\delta)$;
18 $\quad\quad$ **if** $\rho'_{VT} \in \Delta^{\emptyset}_{\rho_{VT}} \cup \Delta^{new}_{\rho_{VT}} \cup \Delta^{mod}_{\rho_{VT}}$ **then**
19 $\quad\quad\quad$ **if** $\rho_{VT} \in \Delta^{\emptyset}_{\rho_{VT}}$ **then**
20 $\quad\quad\quad\quad \Delta^{\emptyset}_{\rho_{VT}} := \Delta^{\emptyset}_{\rho_{VT}} \setminus \{\rho_{VT}\}$;
21 $\quad\quad\quad\quad updateMapping(\triangleleft, \rho_{VT}, \emptyset)$
22 $\quad\quad\quad$ **else if** $\rho_{VT} \in \Delta^{new}_{\rho_{VT}}$ **then**
23 $\quad\quad\quad\quad \Delta^{new}_{\rho_{VT}} := \Delta^{new}_{\rho_{VT}} \setminus \{\rho_{VT}\}$;
24 $\quad\quad\quad$ **else**
25 $\quad\quad\quad\quad \Delta^{mod}_{\rho_{VT}} := \Delta^{mod}_{\rho_{VT}} \setminus \{\rho_{VT}\}$;
26 $\quad\quad\quad\quad updateMapping(\triangleleft, \rho_{VT}, \emptyset)$
27 $\quad\quad$ **else**
28 $\quad\quad\quad$ **if** $\rho_{VT} \in \Delta^{\emptyset}_{\rho_{VT}}$ **then**
29 $\quad\quad\quad\quad \Delta^{\emptyset}_{\rho_{VT}} := \Delta^{\emptyset}_{\rho_{VT}} \setminus \{\rho_{VT}\}$;
30 $\quad\quad\quad\quad \Delta^{mod}_{\rho_{VT}} := \Delta^{mod}_{\rho_{VT}} \cup \{\rho'_{VT}\}$;
31 $\quad\quad\quad\quad updateMapping(\triangleleft, \rho_{VT}, \rho'_{VT})$
32 $\quad\quad\quad$ **else if** $\rho_{VT} \in \Delta^{new}_{\rho_{VT}}$ **then**
33 $\quad\quad\quad\quad \Delta^{new}_{\rho_{VT}} := (\Delta^{new}_{\rho_{VT}} \setminus \{\rho_{VT}\}) \cup \{\rho'_{VT}\}$;
34 $\quad\quad\quad$ **else**
35 $\quad\quad\quad\quad \Delta^{mod}_{\rho_{VT}} := (\Delta^{mod}_{\rho_{VT}} \setminus \{\rho_{VT}\}) \cup \{\rho'_{VT}\}$;
36 $\quad\quad\quad\quad updateMapping(\triangleleft, \rho_{VT}, \rho'_{VT})$
37 **return**;

nodes and the respective update of the derivable variant-tree paths as well as their categorization, we take the addition of delta dependencies and deltas into account for the remaining computation.

Determining New Restrictions. We first incorporate added delta dependencies as they may introduce new restrictions for existing tree paths (cf. Line 9). In contrast to the removal of dependencies and, therefore, restrictions, where new variant-tree paths gets derivable, the addition of restrictions will always reduce the number of tree paths, i.e., some variant-tree paths become invalid and have to be removed from the tree. To determine new restrictions, as shown in Alg. 4.28, we iterate over the added delta dependencies captured in the delta dependency graph regression delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$. For each added dependency *dep*, we identify the source delta δ_{source} as well as the target deltas Δ_{target} of its respective source and target delta nodes and examine the lowest hierarchy level in the variant tree defined by one of the deltas. For some added delta dependencies the incorporation is skipped as the source and target deltas are not yet integrated in the variant tree which is represented by the definition of the hierarchy level as 0. This is the case, when the respective deltas are also added to the delta model and, hence, to the delta dependency graph via the regression delta. The integration of new deltas and their potential restrictions into the variant tree is performed as next computation step. We use the examined hierarchy level to control the remaining determination of potential restrictions such that at least one of the identified deltas has to be contained in the variant tree captured as a hierarchy level. If at least one delta is contained, i.e., $level \neq 0$, we gather the variant tree nodes N^{δ}_{VT} of the identified hierarchy level. Afterwards, we iterate over those tree nodes and compute the variant tree paths starting in the current tree node under consideration up to the root tree node. Based on the path, we determine whether a delta is captured in the path as left or right child tree node which is required to check if the new dependency is valid or not. If the added dependency is invalid, i.e., the combination of deltas defined by the path contradicts the dependency, we identified a new restriction to be handled for the current tree node.

In case a respective delta tree node $n_\delta \in N^{\otimes}_{\Delta}$ of a delta δ is already restricted, the set of restrictions $\alpha(n_\delta)$ is extended. Otherwise, the delta tree node n_δ becomes restricted and is added to the set $N^{\otimes}_{\Delta}$ of restricted delta tree nodes. Accordingly, we remove the variant-tree paths which are not derivable anymore from the set $\Delta^{\theta_{i+1}}_{\rho_{VT}}$ of all tree paths as well as the sets $\Delta^{new}_{\rho_{VT}}$, $\Delta^{mod}_{\rho_{VT}}$, and $\Delta^{\emptyset}_{\rho_{VT}}$ of categorized tree paths. In addition, for all previous variant tree paths that are mapped to those tree paths to be removed via the function $\triangleleft$, we update the mapping such that $\triangleleft$ now points to the empty set. As the left as well as right subtree of n_δ are now invalid, we also remove them from the tree and continue with the next added dependency to be examined.

Integration of New Delta Tree Nodes. Similar to the subtree completion (cf. Alg. 4.23, Line 7), we integrate new deltas by exploiting the standard variant tree computation from Alg. 4.15 such that for each delta a new hierarchy level is added to the variant tree to be recomputed, where delta dependencies are taken into account to determine potential restrictions. As shown in Alg. 4.29, we first determine the set of deltas which were newly added by the higher-order delta $\delta^{H}_{\theta_{i+1}}$ and identify the lowest hierarchy level of the variant tree $VT_{\theta_{i+1}}$ to integrate the new deltas. Afterwards, we iterate over the new deltas and perform the steps of the standard variant tree computation. Hence, we determine for each leaf node whether the path starting in it up to the root node specifies restrictions for the integration of the new delta under the current leaf. Based on the determination, we integrate a delta either as left and/or right child tree node under the leaf and update the variant tree path mapping as shown in Alg. 4.30 before we continue with the next leaf.

Algorithm 4.28.: Function `determineNewRestrictions`

Input: Variant Tree $VT_{\theta_{i+1}}$, Delta Dependency Graph $DG_{DM}^{\theta_{i+1}}$, Delta Dependency Graph Regression Delta $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$, Delta Set $\Delta_{DM_{\theta_{i+1}}}$, Set of New Variant-Tree Paths $\Delta_{\rho_{VT}}^{new}$, Set of Modified Variant-Tree Paths $\Delta_{\rho_{VT}}^{mod}$, Set of Unchanged Variant-Tree Paths $\Delta_{\rho_{VT}}^{\emptyset}$, and Variant-Tree Path Mapping $\triangleleft$

Output: Updated Variant Tree $VT_{\theta_{i+1}}$

1 **Function** *determineNewRestrictions*
2 **forall** $op^{DG_{DM}} \in \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$ **do**
3 **if** $op^{DG_{DM}} ==$ add *dep* **then**
4 $\delta_{source} := getMappedSourceDelta(dep, \Delta_{DM_{\theta_{i+1}}})$;
5 $\Delta_{target} := getMappedTargetDeltas(dep, \Delta_{DM_{\theta_{i+1}}})$;
6 $level := getLowestHierarchyLevel(VT_{\theta_{i+1}}, 0, \delta_{source}, \Delta_{target})$;
7 **if** $level \neq 0$ **then**
8 $N_{VT}^{\delta} := getTreeNodes(VT_{\theta_{i+1}}, level)$;
9 **forall** $n_\delta \in N_{VT}^{\delta}$ **do**
10 $\rho_{VT} := getPath(VT_{\theta_{i+1}}, n_\delta)$;
11 $N_L := N_R := \emptyset$;
12 **forall** $n \in \rho_{VT}$ **do**
13 $n_{DG_{DM}} := getNode(DG_{DM}, \lambda(n))$;
14 **if** $\lambda_{\prec}(n) = L$ **then**
15 $N_L := N_L \cup \{n_{DG_{DM}}\}$;
16 **else**
17 $N_R := N_R \cup \{n_{DG_{DM}}\}$
18 $\delta := getMappedDelta(\Delta_{DM_{\theta_{i+1}}}, n_\delta)$;
19 $n_{DG_{DM}} := getNode(DG_{DM}, \delta)$;
20 **if** $!valid(dep, N_L, N_R, n_\delta)$ **then**
21 **if** $n_\delta \in N_\Delta^{\otimes}$ **then**
22 $\alpha(n_\delta) := \alpha(n_\delta) \cup \{dep\}$;
23 **else**
24 $\Delta_{\rho_{VT}}^{n_\delta} := getPathsWithNode(VT_{\theta_{i+1}}, n_\delta)$;
25 **forall** $\rho_{VT}^{'} \in \Delta_{\rho_{VT}}^{n_\delta}$ **do**
26 **if** $\rho_{VT}^{'} \in \Delta_{\rho_{VT}}^{\emptyset}$ **then**
27 $\Delta_{\rho_{VT}}^{\emptyset} := \Delta_{\rho_{VT}}^{\emptyset} \setminus \{\rho_{VT}^{'}\}$;
28 $updateMapping(\triangleleft, \rho_{VT}^{'}, \emptyset)$
29 **else if** $\rho_{VT}^{'} \in \Delta_{\rho_{VT}}^{new}$ **then**
30 $\Delta_{\rho_{VT}}^{new} := \Delta_{\rho_{VT}}^{new} \setminus \{\rho_{VT}^{'}\}$;
31 **else**
32 $\Delta_{\rho_{VT}}^{mod} := \Delta_{\rho_{VT}}^{mod} \setminus \{\rho_{VT}^{'}\}$;
33 $updateMapping(\triangleleft, \rho_{VT}^{'}, \emptyset)$
34 $removeSubtrees(VT_{\theta_{i+1}}, n_\delta)$;
35 $N_\Delta := N_\Delta \setminus \{n_\delta\}$;
36 $N_\Delta^{\otimes} := N_\Delta^{\otimes} \cup \{n_\delta\}$;
37 $\alpha(n_\delta) := \{dep\}$;
38 $\prec_L (n_\delta) := \prec_R (n_\delta) := \otimes$
39 **return** $VT_{\theta_{i+1}}$;

Algorithm 4.29.: Function `integrateNewDeltas`

Input: Variant Tree $VT_{\theta_{i+1}}$, Delta Dependency Graph $DG_{DM}^{\theta_{i+1}}$, Higher-Order Delta $\delta_{\theta_{i+1}}^{H}$, Delta Set $\Delta_{DM_{\theta_{i+1}}}$, Set of New Variant-Tree Paths $\Delta_{\rho_{VT}}^{new}$, Set of Modified Variant-Tree Paths $\Delta_{\rho_{VT}}^{mod}$, Set of Unchanged Variant-Tree Paths $\Delta_{\rho_{VT}}^{\emptyset}$, and Variant-Tree Path Mapping $\triangleleft$

Output: Variant Tree $VT_{\theta_{i+1}}$

1 **Function** *integrateNewDeltas*
2 $\quad \Delta_{new} := \emptyset$;
3 $\quad$ **forall** $op^H \in \delta_{\theta_{i+1}}^{H}$ **do**
4 $\quad\quad$ **if** $op^H ==$ add δ **then**
5 $\quad\quad\quad \Delta_{new} := \Delta_{new} \cup \{\delta\}$;
6 $\quad level := getLowestHierarchyLevel(VT_{\theta_{i+1}})$;
7 $\quad$ **forall** $\delta \in \Delta_{new}$ **do**
8 $\quad\quad N_{\Delta}^{level} := getLeafs(VT_{\theta_{i+1}}, level)$;
9 $\quad\quad$ **forall** $n_{\delta'} \in N_{\Delta}^{level}$ **do**
10 $\quad\quad\quad Dep_{DM}^{\delta,L} := \emptyset$;
11 $\quad\quad\quad Dep_{DM}^{\delta,R} := \emptyset$;
12 $\quad\quad\quad \rho_{VT} := getPath(VT_{\theta_{i+1}}, n_{\delta'})$;
13 $\quad\quad\quad Dep_{DM}^{\delta,L} := checkDepForRestriction(VT_{\theta_{i+1}}, DG_{DM}^{\theta_{i+1}}, \rho_{VT}, \delta,$ `LEFT`$)$;
14 $\quad\quad\quad Dep_{DM}^{\delta,R} := checkDepForRestriction(VT_{\theta_{i+1}}, DG_{DM}^{\theta_{i+1}}, \rho_{VT}, \delta,$ `RIGHT`$)$;
15 $\quad\quad\quad$ **if** $Dep_{DM}^{\delta,L} == \emptyset \wedge Dep_{DM}^{\delta,R} == \emptyset$ **then**
16 $\quad\quad\quad\quad VT_{\theta_{i+1}} := addChild(VT_{\theta_{i+1}}, n_{\delta'},$ `LEFT_RIGHT`$, \delta, \emptyset)$;
17 $\quad\quad\quad$ **else if** $Dep_{DM}^{\delta,L} == \emptyset \wedge Dep_{DM}^{\delta,R} \neq \emptyset$ **then**
18 $\quad\quad\quad\quad VT_{\theta_{i+1}} := addChild(VT_{\theta_{i+1}}, n_{\delta'},$ `LEFT`$, \delta, Dep_{DM}^{\delta})$;
19 $\quad\quad\quad$ **else if** $Dep_{DM}^{\delta,L} \neq \emptyset \wedge Dep_{DM}^{\delta,R} == \emptyset$ **then**
20 $\quad\quad\quad\quad VT_{\theta_{i+1}} := addChild(VT_{\theta_{i+1}}, n_{\delta'},$ `RIGHT`$, \delta, Dep_{DM}^{\delta})$;
21 $\quad\quad\quad updatePathMappingAfterAddition(\rho_{VT}, \prec_L (n_{\delta'}), \prec_R (n_{\delta'}), \Delta_{\rho_{VT}}^{new}, \Delta_{\rho_{VT}}^{mod}, \Delta_{\rho_{VT}}^{\emptyset}, \triangleleft)$;
22 $\quad\quad level$++;
23 **return** $VT_{\theta_{i+1}}$;

The addition of a delta tree node to the variant tree either (1) as left child tree node, (2) as right child tree node, or (3) as left and right child tree node has an impact on the set of derivable variant-tree paths and their preliminary categorization. In case a delta tree node n_δ is integrated by extending an existing tree path $\rho_{VT}^{'} = ((\ldots, n_\delta^L), \lambda_\prec)$ solely as left child, the variant which was represented by the previous variant-tree path and, hence, by the respective delta set does not exist anymore. If the previous variant-tree path was categorized as new or modified, the respective categorization is still valid for the extended path. In contrast, a previously unchanged tree path is now classified as modified. In case a delta tree node n_δ is integrated by extending an existing tree path $\rho_{VT}^{'} = ((\ldots, n_\delta^R), \lambda_\prec)$ solely as right child, the variant which was represented by the previous variant-tree path does still exist as the new delta is not selected for the respective delta set. Hence, the preliminary categorization as new, modified, and unchanged is still valid and nothing changes. In case a delta tree node n_δ is integrated as left and right child, there is a new variant-tree path $\rho_{VT}^{'} = ((\ldots, n_\delta^L), \lambda_\prec)$ derivable in addition to the existing path $\rho_{VT}^{''} = ((\ldots, n_\delta^R), \lambda_\prec)$. As the pre-

Algorithm 4.30.: Function `updatePathMappingAfterAddition`

Input: Adaptable Variant-Tree Path ρ_{VT}, Left-Child Tree Node n_{δ}^{L}, Right-Child Tree Node n_{δ}^{R}, Set of New Variant-Tree Paths $\Delta_{\rho_{VT}}^{new}$, Set of Modified Variant-Tree Paths $\Delta_{\rho_{VT}}^{mod}$, Set of Unchanged Variant-Tree Paths $\Delta_{\rho_{VT}}^{\varnothing}$, and Variant-Tree Path Mapping $\triangleleft$

1 **Function** *updatePathMappingAfterAddition*
2 **if** $n_{\delta}^{L} \notin N_{\Delta}^{\otimes} \wedge n_{\delta}^{R} \notin N_{\Delta}^{\otimes}$ **then**
3 $\rho_{VT}^{'} := addLeftNodeToSequence(\rho_{VT}, n_{\delta}^{L})$;
4 $\rho_{VT}^{''} := addRightNodeToSequence(\rho_{VT}, n_{\delta}^{R})$;
5 **if** $\rho_{VT} \in \Delta_{\rho_{VT}}^{\varnothing}$ **then**
6 $\Delta_{\rho_{VT}}^{\varnothing} := (\Delta_{\rho_{VT}}^{\varnothing} \setminus \{\rho_{VT}\}) \cup \{\rho_{VT}^{''}\}$;
7 $updateMapping(\triangleleft, \rho_{VT}, \rho_{VT}^{''})$;
8 $\Delta_{\rho_{VT}}^{new} := \Delta_{\rho_{VT}}^{new} \cup \{\rho_{VT}^{'}\}$;
9 **else if** $\rho_{VT} \in \Delta_{\rho_{VT}}^{new}$ **then**
10 $\Delta_{\rho_{VT}}^{new} := (\Delta_{\rho_{VT}}^{new} \setminus \{\rho_{VT}\}) \cup \{\rho_{VT}^{'}, \rho_{VT}^{''}\}$;
11 **else**
12 $\Delta_{\rho_{VT}}^{mod} := (\Delta_{\rho_{VT}}^{mod} \setminus \{\rho_{VT}\}) \cup \{\rho_{VT}^{''}\}$;
13 $updateMapping(\triangleleft, \rho_{VT}, \rho_{VT}^{''})$;
14 $\Delta_{\rho_{VT}}^{new} := \Delta_{\rho_{VT}}^{new} \cup \{\rho_{VT}^{'}\}$;
15 **else if** $n_{\delta}^{L} \notin N_{\Delta}^{\otimes} \wedge n_{\delta}^{R} \in N_{\Delta}^{\otimes}$ **then**
16 $\rho_{VT}^{'} := addLeftNodeToSequence(\rho_{VT}, n_{\delta}^{L})$;
17 **if** $\rho_{VT} \in \Delta_{\rho_{VT}}^{\varnothing}$ **then**
18 $\Delta_{\rho_{VT}}^{\varnothing} := \Delta_{\rho_{VT}}^{\varnothing} \setminus \{\rho_{VT}\}$;
19 $updateMapping(\triangleleft, \rho_{VT}, \rho_{VT}^{'})$;
20 $\Delta_{\rho_{VT}}^{mod} := \Delta_{\rho_{VT}}^{mod} \cup \{\rho_{VT}^{'}\}$;
21 **else if** $\rho_{VT} \in \Delta_{\rho_{VT}}^{new}$ **then**
22 $\Delta_{\rho_{VT}}^{new} := (\Delta_{\rho_{VT}}^{new} \setminus \{\rho_{VT}\}) \cup \{\rho_{VT}^{'}\}$;
23 **else**
24 $\Delta_{\rho_{VT}}^{mod} := (\Delta_{\rho_{VT}}^{mod} \setminus \{\rho_{VT}\}) \cup \{\rho_{VT}^{'}\}$;
25 $updateMapping(\triangleleft, \rho_{VT}, \rho_{VT}^{'})$;
26 **else if** $n_{\delta}^{L} \in N_{\Delta}^{\otimes} \wedge n_{\delta}^{R} \notin N_{\Delta}^{\otimes}$ **then**
27 $\rho_{VT}^{'} := addRightNodeToSequence(\rho_{VT}, n_{\delta}^{R})$;
28 **if** $\rho_{VT} \in \Delta_{\rho_{VT}}^{\varnothing}$ **then**
29 $\Delta_{\rho_{VT}}^{\varnothing} := (\Delta_{\rho_{VT}}^{\varnothing} \setminus \{\rho_{VT}\}) \cup \{\rho_{VT}^{'}\}$;
30 $updateMapping(\triangleleft, \rho_{VT}, \rho_{VT}^{'})$;
31 **else if** $\rho_{VT} \in \Delta_{\rho_{VT}}^{new}$ **then**
32 $\Delta_{\rho_{VT}}^{new} := (\Delta_{\rho_{VT}}^{new} \setminus \{\rho_{VT}\}) \cup \{\rho_{VT}^{'}\}$;
33 **else**
34 $\Delta_{\rho_{VT}}^{mod} := (\Delta_{\rho_{VT}}^{mod} \setminus \{\rho_{VT}\}) \cup \{\rho_{VT}^{'}\}$;
35 $updateMapping(\triangleleft, \rho_{VT}, \rho_{VT}^{'})$;
36 **return**;

Algorithm 4.31.: Function `incorporateModDeltas`

Input: Variant Tree $VT_{\theta_{i+1}}$, Higher-Order Delta $\delta^H_{\theta_{i+1}}$, Set of Modified Variant-Tree Paths $\Delta^{mod}_{\rho_{VT}}$, Set of Unchanged Variant-Tree Paths $\Delta^{\emptyset}_{\rho_{VT}}$, and Variant-Tree Path Mapping $\triangleleft$

```
1  Function incorporateModDeltas
2      forall op^H ∈ δ^H_{θ_{i+1}} do
3          if op^H == mod(δ, {add op, rem op', ...}) ∨ op^H == mod(δ, r'_ε) then
4              forall ρ_VT ∈ Δ^∅_{ρ_VT} do
5                  if ρ_VT == ((..., n^L_δ, ...), λ_≺) then
6                      Δ^∅_{ρ_VT} := Δ^∅_{ρ_VT} \ {ρ_VT};
7                      Δ^mod_{ρ_VT} := Δ^mod_{ρ_VT} ∪ {ρ_VT};
8  return;
```

vious tree path $\rho''_{VT} = ((\ldots, n^R_\delta), \lambda_\prec)$ still exists, its current categorization is valid. Accordingly, we add the new derivable variant-tree path $\rho'_{VT} = ((\ldots, n^L_\delta), \lambda_\prec)$ to the set $\Delta^{new}_{\rho_{VT}}$. For all three cases, we update the mapping of the previous variant-tree paths such that the function $\triangleleft$ now points to the extended tree paths, whereas the newly derivable tree paths are not contained in any mapping to a previous path. After finishing the integration of new deltas, we have to incorporate the potential modification of deltas in the categorization as the last step of the incremental variant tree computation.

Incorporation of Delta Modifications. By taking the modification of deltas captured in the higher-order delta $\delta^H_{\theta_{i+1}}$ into account, the set of derivable variant-tree paths as well as the variant-tree path mapping $\triangleleft$ do not change, but the categorization of unchanged tree paths may be affected (cf. Line 11). As described in Sect. 3.3, a delta δ is modified by (1) altering its application condition φ_δ, (2) exchanging the region r_ϵ the encapsulated change operations OP_δ are applied to, or (3) altering the set of encapsulated change operation OP_δ itself. In the first case, the modification is already handled as a change of the application condition results in different delta dependencies. Therefore, as shown in Alg. 4.31, we focus on the other two cases, where the internal of a delta, i.e., its set of encapsulated change operations, is modified. We determine those variant-tree paths $\rho_{VT} = ((\ldots, n^L_\delta, \ldots), \lambda_\prec)$ that contain the modified delta as left child node and are categorized as unchanged. For those tree paths, we solely have to update the categorization to modified by adding them to the set $\Delta^{mod}_{\rho_{VT}}$.

In the end, the incremental computation of the variant tree terminates since the change operations captured in the delta dependency graph regression delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$ as well as the higher-order delta $\delta^H_{\theta_{i+1}}$, and the set of delta tree nodes N_{VT} are finite. As result of the computation, we obtain the valid variant tree $VT_{\theta_{i+1}} = \Delta^{\emptyset}_{\rho_{VT}} \cup \Delta^{new}_{\rho_{VT}} \cup \Delta^{mod}_{\rho_{VT}}$ of the current SPL version θ_{i+1}, the categorization of the derivable tree paths by means of new $\Delta^{new}_{\rho_{VT}}$, modified $\Delta^{mod}_{\rho_{VT}}$, and unchanged $\Delta^{\emptyset}_{\rho_{VT}}$ variant tree paths, and also the final mapping $\triangleleft$ between previous variant-tree paths $\Delta^{\theta_i}_{\rho_{VT}}$ and the adapted set $\Delta^{\theta_{i+1}}_{\rho_{VT}}$. The categorization as well as the mapping facilitate the reasoning about the higher-order delta application such that we are able to derive the variant set evolution delta $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$.

Variant Set Evolution Delta Derivation. Each variant-tree path $\rho^v_{VT} \in \Delta^{\theta_{i+1}}_{\rho_{VT}}$ represents the delta set Δ_v of a certain variant $v \in \mathbb{V}_{\theta_{i+1}}$ of SPL version θ_{i+1}. For the derivation of the variant set evolution delta $\Delta^{SM_{\mathbb{V}}}_{\theta_i,\theta_{i+1}}$, we iterate over all variant-tree paths $\rho^v_{VT} \in \Delta^{\theta_{i+1}}_{\rho_{VT}}$ to derive respective variant set evolution

change operations $op^{\mathcal{SM}}$ by taking the categorization as new, modified, and unchanged tree paths as well as the mapping to previous variant-tree paths $\Delta^{\theta_i}_{\rho_{VT}}$ into account. First, we determine the delta set Δ_v from a tree path ρ^v_{VT} by selecting those deltas $\delta \in \Delta_{DM_{\theta_{i+1}}}$ which are mapped to left-child delta tree nodes. Second, we apply the delta set to the core state machine $sm_{v_{core}}$ of the delta model $DM_{\theta_{i+1}}$ to obtain the variant-specific state machine $sm_v = \mathsf{apply}_\delta(sm_{v_{core}}, \Delta_v)$. Third, we use the categorization and the mapping $\triangleleft$ to derive the type of the variant set evolution change operation.

- For new tree paths $\rho^v_{VT} \in \Delta^{new}_{\rho_{VT}}$, we integrate an add operation $(\mathsf{add}\ sm_v)$ of state machine sm_v into the variant set evolution delta $\Delta^{SM_V}_{\theta_i,\theta_{i+1}}$.
- For modified tree paths $\rho^v_{VT} \in \Delta^{mod}_{\rho_{VT}}$, we integrate a modify operation $(\mathsf{mod}\ (sm_{v'}, sm_v))$ from state machine $sm_{v'}$ to state machine sm_v into $\Delta^{SM_V}_{\theta_i,\theta_{i+1}}$, where $sm_{v'}$ represents a previous state machine of SPL version θ_i. To obtain $sm_{v'}$, we examine the mapping function $\triangleleft$ to find the respective previous tree path $\rho^{v'}_{VT}$, use the delta model DM_{θ_i} of the previous SPL version θ_i to determine the delta set $\Delta_{v'}$ and apply this delta set to the core state machine.
- We perform similar steps to integrate a remove operation $(\mathsf{rem}\ sm_v)$ of a state machine sm_v of the previous SPL version θ_i into $\Delta^{SM_V}_{\theta_i,\theta_{i+1}}$. Again, we examine the mapping function $\triangleleft$ to find those previous tree paths $\rho^v_{VT} \in \Delta^{\theta_i}_{\rho_{VT}}$ that are mapped $\triangleleft(\rho^v_{VT}) = \emptyset$ to the empty set, determine their delta sets Δ_v and apply them to the core to obtain the state machine sm_v to be removed.
- For unchanged tree paths $\rho^v_{VT} \in \Delta^{\emptyset}_{\rho_{VT}}$, we do not integrate change operations as the variants are identical in both SPL versions and are not affected by the higher-order delta application.

In the end, the variant set evolution delta $\Delta^{SM_V}_{\theta_i,\theta_{i+1}}$ captures the impact of the higher-order delta application in terms of added, removed, and modified variants. We exploit the evolution delta $\Delta^{SM_V}_{\theta_i,\theta_{i+1}}$ and also the variant-tree path mapping $\triangleleft$ to guide the retest of versions of variants in our model-based regression testing framework when stepping to the next SPL version to be tested as described in the next chapter.

Table 4.5: Symbol Summary of Incremental Delta Set Derivation Definition

Symbol	Description
$op^{DG_{DM}}, \mathcal{OP}^{DG_{DM}}$	Delta dependency graph change operation; Universe of delta dependency graph change operation
$\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$	Delta dependency graph regression delta
$\mathsf{apply}_{DG_{DM}}$	Delta dependency graph delta application function
$\triangleleft$	Variant-tree path mapping function

In Tab. 4.5, we summarize the list of symbols used for the definition of the incremental delta set derivation. To recapitulate, when stepping to the next SPL version θ_{i+1}, we first adapt the delta dependency graph similar to the dependency graph adaptation of our slicing technique. We determine delta dependency graph change operations $op^{DG_{DM}}$ specifying the addition or removal of delta nodes and delta dependency edges by exploiting the evolution change operations of the applied higher-order delta $\delta^H_{\theta_{i+1}}$. We capture delta dependency graph change operations in a delta dependency graph regression delta $\Delta^{DG_{DM}}_{\theta_i,\theta_{i+1}}$ such that we obtain the delta dependency graph $DG^{\theta_{i+1}}_{DM} =$

$\mathsf{apply}_{DG_{DM}}(DG_{DM}^{\theta_i}, \Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}})$ of SPL version θ_{i+1} by applying the delta dependency graph regression delta $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$ to the delta dependency graph $DG_{DM}^{\theta_i}$ of the previous SPL version θ_i via the application function $\mathsf{apply}_{DG_{DM}}$. Afterwards, we incorporate the change operations of the regression delta $\Delta_{\theta_i,\theta_{i+1}}^{DG_{DM}}$ and the higher-order delta $\delta_{\theta_{i+1}}^{H}$ to incrementally recompute the variant tree $VT_{\theta_{i+1}}$ based on its previous version VT_{θ_i}. During the recomputation, we categorize the derivable variant-tree paths as new $\Delta_{\rho_{VT}}^{new}$, modified $\Delta_{\rho_{VT}}^{mod}$, or unchanged $\Delta_{\rho_{VT}}^{\emptyset}$ and further ensure a mapping between previous tree paths $\Delta_{\rho_{VT}}^{\theta_i}$ of VT_{θ_i} and current paths $\Delta_{\rho_{VT}}^{\theta_{i+1}}$ of $VT_{\theta_{i+1}}$ via the mapping function $\triangleleft$. In the end, both the categorization and the mapping are used to derive the variant set evolution delta $\Delta_{\theta_i,\theta_{i+1}}^{SM_{\mathbb{V}}}$.

Example 4.8: Incremental Delta Set Derivation

Consider the variant tree VT_{θ_1} of SPL version θ_1 depicted in Fig. 4.11a. Based on the change operations of the higher-order delta $\delta_{\theta_1}^{H}$ from Ex. 3.5 and the delta dependency graph regression delta $\Delta_{\theta_0,\theta_1}^{DG_{DM}}$ from Ex. 4.7, we recompute the variant tree such that we solely have to add the hierarchy level for the new delta δ_6. Therefore, the variant tree VT_{θ_1} differs to the variant tree VT_{θ_0} from Ex. 4.6 shown Fig. 4.9 solely in this hierarchy level. Thats why, we abstract from the identical restrictions in Fig. 4.11a and only provides the new restrictions introduced based on the addition of δ_6. In addition, we represent the modification of δ_1 to δ_1' via dashed lines in Fig. 4.11a. The variant tree VT_{θ_1} allows for the derivation of the five paths

- $\rho_{VT}^{v'_{core}} = ((n_{\delta_6}^L, n_{\delta_5}^R, n_{\delta_4}^R, n_{\delta_3}^R, n_{\delta_2}^R, n_{\delta_1'}^R), \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = \lambda_{\prec}(n_{\delta_3}) = \lambda_{\prec}(n_2) = \lambda_{\prec}(n_{\delta_1'}) = R; \lambda_{\prec}(n_{\delta_6}) = L)$,
- $\rho_{VT}^{v'_1} = ((n_{\delta_6}^L, n_{\delta_5}^R, n_{\delta_4}^R, n_{\delta_3}^L, n_{\delta_2}^L, n_{\delta_1'}^L), \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = R; \lambda_{\prec}(n_{\delta_6}) = \lambda_{\prec}(n_{\delta_3}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta_1'}) = L)$,
- $\rho_{VT}^{v'_2} = ((n_{\delta_6}^L, n_{\delta_5}^R, n_{\delta_4}^L, n_{\delta_3}^R, n_{\delta_2}^R, n_{\delta_1'}^R), \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_3}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta_1'}) = R; \lambda_{\prec}(n_{\delta_6}) = \lambda_{\prec}(n_{\delta_4}) = L)$,
- $\rho_{VT}^{v'_3} = ((n_{\delta_6}^R, n_{\delta_5}^L, n_{\delta_4}^L, n_{\delta_3}^R, n_{\delta_2}^L, n_{\delta_1'}^L), \lambda_{\prec}(n_{\delta_6}) = \lambda_{\prec}(n_{\delta_3}) = R; \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta_1'}) = L)$, and
- $\rho_{VT}^{v_4} = ((n_{\delta_6}^L, n_{\delta_5}^L, n_{\delta_4}^L, n_{\delta_3}^R, n_{\delta_2}^L, n_{\delta_1'}^L), \lambda_{\prec}(n_{\delta_3}) = R; \lambda_{\prec}(n_{\delta_6}) = \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta_1'}) = L)$.

We determine the following mapping to previous variant-tree paths $\Delta_{\rho_{VT}}^{\theta_0}$ from Ex. 4.6

- $\triangleleft(\rho_{VT}^{v_{core}}) = \rho_{VT}^{v'_{core}}$,
- $\triangleleft(\rho_{VT}^{v_1}) = \rho_{VT}^{v'_1}$,
- $\triangleleft(\rho_{VT}^{v_2}) = \rho_{VT}^{v'_2}$, and
- $\triangleleft(\rho_{VT}^{v_3}) = \rho_{VT}^{v'_3}$.

Furthermore, we obtain the following categorization

- $\Delta_{\rho_{VT}}^{new} = \{\rho_{VT}^{v_4}\}$,
- $\Delta_{\rho_{VT}}^{mod} = \{\rho_{VT}^{v'_{core}}, \rho_{VT}^{v'_1}, \rho_{VT}^{v'_2}, \rho_{VT}^{v'_3}\}$, and
- $\Delta_{\rho_{VT}}^{\emptyset} = \emptyset$.

Based on the mapping and the categorization, the variant set evolution delta $\Delta_{\theta_0,\theta_1}^{SM_{\mathbb{V}}}$ between SPL version θ_0 and θ_1 representing the impact of higher-order delta $\delta_{\theta_1}^{H}$ is defined by $\Delta_{\theta_0,\theta_1}^{SM_{\mathbb{V}}} =$

$\{\mathsf{mod}\,(sm_{v_{core}}, sm_{v'_{core}}), \mathsf{mod}\,(sm_{v_1}, sm_{v'_1}), \mathsf{mod}\,(sm_{v_2}, sm_{v'_2}), \mathsf{mod}\,(sm_{v_3}, sm_{v'_3}), \mathsf{add}\,sm_{v_4}\}$, i.e., we modify variants v_{core}, v_1, v_2, and v_3 as well as add the new variant v_4.

When stepping to SPL version θ_2 by applying the empty higher-order delta $\delta^H_{\theta_2}$ (cf. Ex. 3.5), its impact to the variant tree VT_{θ_2} is depicted in Fig. 4.11b. Based on the change operations of the delta dependency graph regression delta $\Delta^{DG_{DM}}_{\theta_1,\theta_2}$ from Ex. 4.7, we recompute the variant tree such that we have to exchange the restrictions of the far right delta tree node of the last hierarchy level and also remove the single restriction of the second delta tree node from the left of the last hierarchy level. Due to the removal of the single restriction, a new variant-tree path $\rho^{v_5}_{VT} = ((n^R_{\delta_6}, n^R_{\delta_5}, n^R_{\delta_4}, n^L_{\delta_3}, n^L_{\delta_2}, n^L_{\delta'_1}), \lambda_{\prec}(n_{\delta_6}) = \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = R; \lambda_{\prec}(n_{\delta_3}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta_1}) = L)$ is derivable. We determine the following mapping to previous variant-tree paths $\Delta^{\theta_1}_{\rho_{VT}}$

- $\lhd(\rho^{v_{core}}_{VT}) = \rho^{v_{core}}_{VT}$,
- $\lhd(\rho^{v_1}_{VT}) = \rho^{v_1}_{VT}$,
- $\lhd(\rho^{v_2}_{VT}) = \rho^{v_2}_{VT}$,
- $\lhd(\rho^{v_3}_{VT}) = \rho^{v_3}_{VT}$, and
- $\lhd(\rho^{v_4}_{VT}) = \rho^{v_4}_{VT}$.

In addition, we obtain the following categorization

- $\Delta^{new}_{\rho_{VT}} = \{\rho^{v_5}_{VT}\}$,
- $\Delta^{mod}_{\rho_{VT}} = \emptyset$, and
- $\Delta^{\emptyset}_{\rho_{VT}} = \{\rho^{v_{core}}_{VT}, \rho^{v_1}_{VT}, \rho^{v_2}_{VT}, \rho^{v_3}_{VT}, \rho^{v_4}_{VT}\}$.

Therefore, the variant set evolution delta $\Delta^{SM_V}_{\theta_1,\theta_2}$ between SPL version θ_1 and θ_2 representing the impact of higher-order delta $\delta^H_{\theta_2}$ is defined by $\Delta^{SM_V}_{\theta_1,\theta_2} = \{\mathsf{add}\,sm_{v_5}\}$, i.e., we solely add the new variant v_5 as the remaining four variants are unchanged.

4.3 Implementation and Evaluation

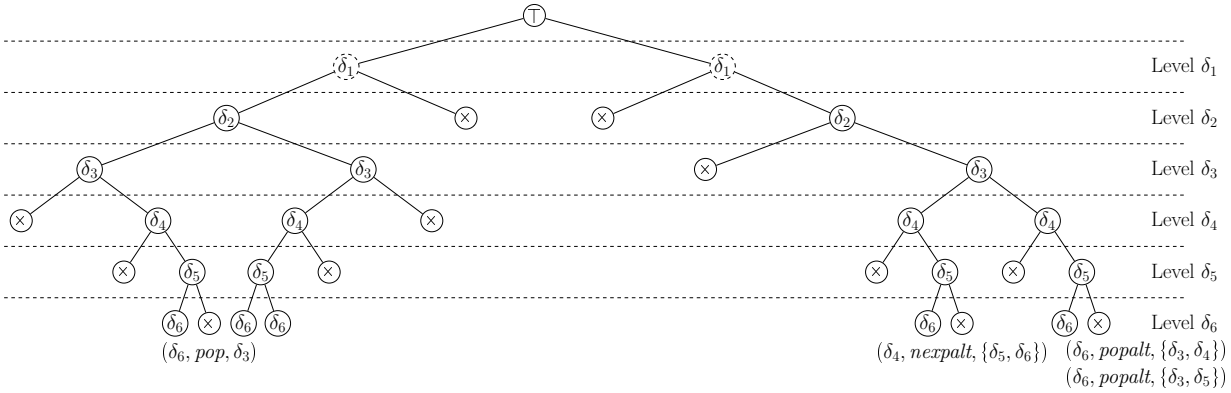
In this section, we shortly present our prototypical implementation facilitating the change impact analyses of variants and versions of variants. Furthermore, we describe the evaluation of our incremental slicing technique and of the reasoning about higher-order delta application, where we use the three evolving delta-oriented product lines introduced in Chapt. 3 as subject systems.

4.3.1 Prototype

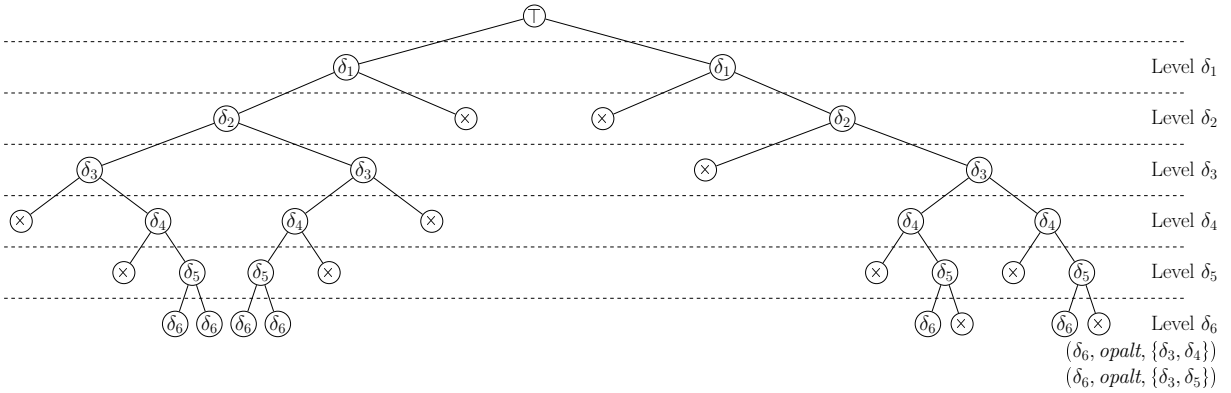
For our change impact analysis techniques, we provide respective prototypical implementations which are also realized as Eclipse[1] plug-ins using EMF.[2] In particular, the following plug-ins are specified for our incremental model slicing and the reasoning about higher-order delta application based on corresponding meta models:

- `de.imotep.slicing.dependency` – Plug-in for the slicing dependency analysis (cf. Sect. 4.1.1).
- `de.imotep.slicing` – Plug-in for incremental model slicing (cf. Sect. 4.1.2).
- `de.imotep.dope.dependency` – Plug-in for the reasoning about higher-order delta application (cf. Sect. 4.2).

[1] https://www.eclipse.org/, last access: May 31st, 2019
[2] https://www.eclipse.org/modeling/emf/, last access: May 31st, 2019



(a) Variant Tree VT_{θ_1} for the Delta Model DM_{θ_1} of SPL Version θ_1



(b) Variant Tree VT_{θ_2} for the Delta Model DM_{θ_2} of SPL Version θ_2

Figure 4.11: Variant Trees VT_{θ_1} and VT_{θ_2} for SPL Versions θ_1 and θ_2

Similar to the plug-ins for delta-oriented test modeling (cf. Sect. 3.4), the plug-ins facilitating the distinct change impact analyses are part of the tool support of the research project IMoTEP.[3] The meta models and, therefore, the plug-ins allow for the improvements and extensions of our change impact anlysis techniques in the future, e.g., by incorporating further control or data dependencies for the incremental slicing technique. In the following paragraphs, we show and describe the meta models of the plug-ins.

Slicing Dependency Analysis. We require the plug-in `de.imotep.slicing.dependency` for the control dependency analysis captured in a dependency graph as well as for the incremental dependency graph adaptation used during the incremental slicing process as described in Sect. 4.1. For a better illustration, we split the meta model in two parts. In Fig. 4.12, the main part of the meta model is shown, whereas in Fig. 4.13 the delta-oriented part is depicted. The main class of the meta model shown in Fig. 4.12 is `DependencyGraph`. A `DependencyGraph` comprises a set of `ElementNodes`, a set of `Dependencys`, and a set of `DependencyEdges`. In addition, a `DependencyGraph` has a reference to a `StateMachine` for which the dependency analysis is to be performed. An `ElementNode` is either a `StateNode` or a `TransitionNode` as we solely focus on states and transitions during slicing, where a `StateNode` is mapped to a `State` which it represents and a `TransitionNode` is mapped to a `Transition`, respectively. The four control dependencies applied in this thesis are realized

[3] http://www.dfg-spp1593.de/imotep/, last access: May 31st, 2019

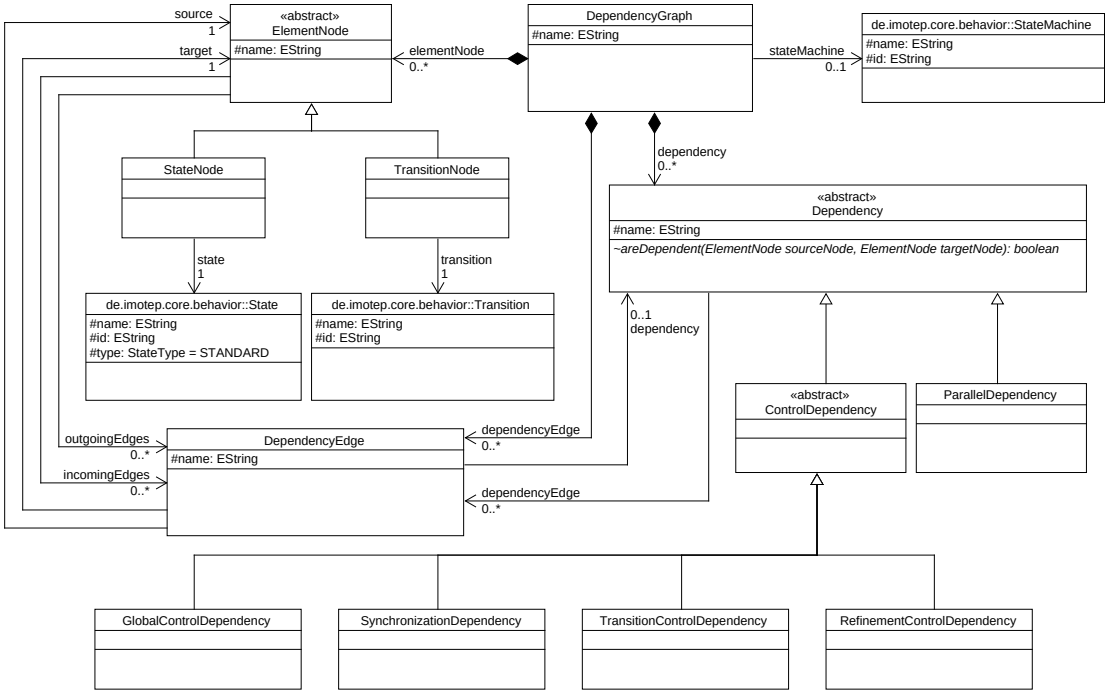


Figure 4.12: Meta Model of the Slicing Dependency Analysis Plug-In (Main Part)

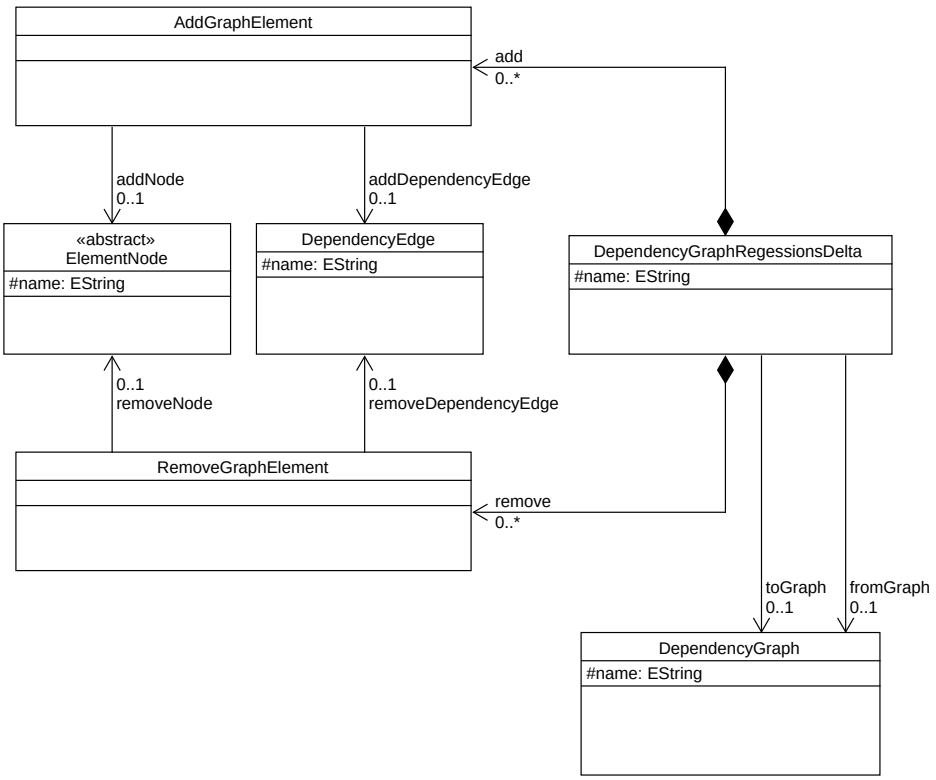


Figure 4.13: Meta Model of the Slicing Dependency Analysis Plug-In (Delta-Oriented Part)

by the classes GlobalControlDependency, SynchronizationDependency, TransitionControlDependency, and RefinementControlDependency. They inherit from the abstract class ControlDependency which in turn inherits from Dependency. We integrated the intermediate class ControlDependency to facilitate a lightweight integration of data dependencies in the future, e.g., by adding an intermediate class DataDependency from which the real data dependencies can inherit. In addition to ControlDependency, we use ParallelDependency as auxiliary dependency improving the computation of parallel control dependencies such as SynchronizationDependency. We determine whether two ElementNodes are dependent via the abstract method areDependent() which is implemented accordingly for each control dependency in the respective class. To specify that two ElementNodes and, therefore, their represented state machine elements are dependent, a DependencyEdge connects both ElementNodes, where one ElementNode serves as source and the other ElementNode as target. As an ElementNode is potentially dependent to several ElementNodes represented by the connections via DependencyEdge, we relate each ElementNode to its set of incomingEdges as well as outgoingEdges. In general, a DependencyEdge has no direct type. Hence, we map the edge to a Dependency for which it is created between two ElementNodes. In return, every Dependency is mapped to a set of DependencyEdges contained in the DependencyGraph.

For the incremental dependency graph adaptation, the main class is DependencyGraphRegressionDelta as shown in Fig. 4.13. A DependencyGraphRegressionDelta is mapped to the DependencyGraph which is transformed by the regression delta via fromGraph and also to the resulting DependencyGraph via toGraph. In addition, a DependencyGraphRegressionDelta comprises the dependency graph change operations realized as AddGraphElement for additions and RemoveGraphElement for removals. An AddGraphElement as well as a RemoveGraphElement is mapped to the ElementNode and DependencyEdge it adds or removes, respectively.

Incremental Model Slicing. The plug-in de.imotep.slicing implements our incremental model slicing technique which is applied for change impact analysis between variants and version of variants as described in Sect. 4.1. Again, we split the meta model in two parts to allow for a better illustration. In Fig. 4.14, the main part of the meta model is depicted, whereas in Fig. 4.15 the delta-oriented part is shown. The main class of the meta model shown in Fig. 4.14 is SlicingManager. A SlicingManager controls the creation and adaptation of a DependencyGraph and a Slice via the methods (1) buildDependencyGraph() as well as createSlice() for the standard slicing, and (2) incrementalDependencyGraphAdaptation() as well as incrementalSliceComputation() for the incremental slicing. For the application of slicing, we have to define a SlicingStrategy. The class SlicingStrategy is abstract and the class SlicingStrategyBackward inherits from it. Similar as the abstract intermediate class ControlDependency in the plug-in de.imotep.slicing.dependency, we use SlicingStrategy as an intermediate class to facilitate a lightweight integration of forward slicing in our slicing technique in the future. Each SlicingStrategy is mapped to the current DependencyGraph used for the analysis. All DependencyGraphs which are created during the slicing application are captured by the SlicingManager. If a DependencyGraph is generated incrementally, the SlicingManager records the derived DependencyGraphRegressionDelta.

To compute a Slice of a given StateMachine which is referenced as original, we require the DepedendencyGraph as well as a SlicingCriterion. A SlicingCriterion defines a SliceGoal which is either a SliceStateGoal or SliceTransitionGoal. Both classes have a reference to the StateMachine element State or Transition, respectively. As described in Sect 4.1.1, we solely

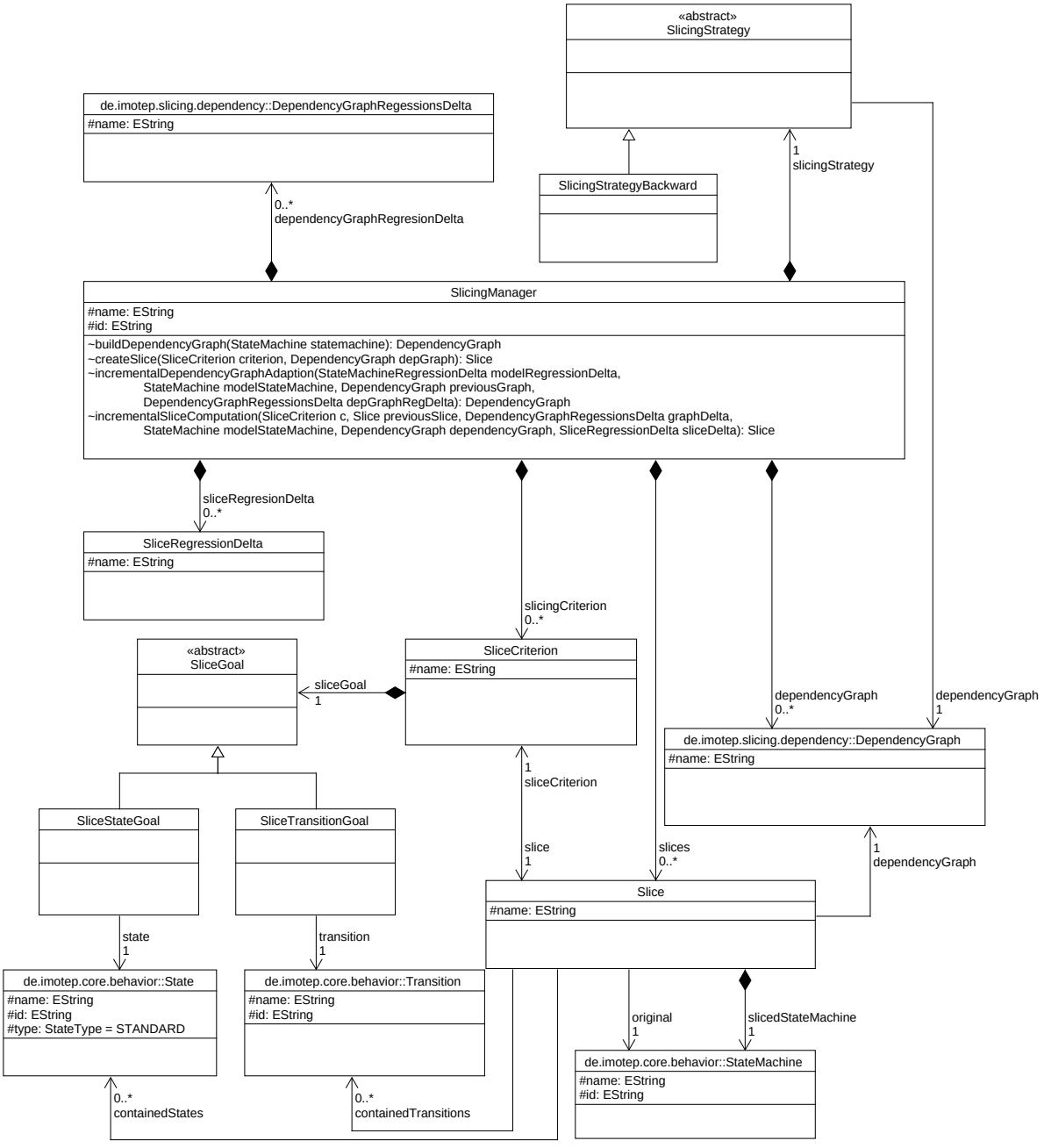


Figure 4.14: Meta Model of the Incremental Model Slicing Plug-In (Main Part)

focus on states and transitions during the slice computation. Therefore, a `Slice` is mapped to the sets of `States` and `Transitions` it contains. If required, we are able to compute the `slicedStateMachine` for a `Slice` based on its containing `States` as well as `Transitions`. All created `Slices` and `SliceCriterions` are captured by the `SlicingManager`.

If a `Slice` is computed incrementally, the `SlicingManager` records the `SliceRegressionDelta`. As depicted in Fig. 4.15, a `SliceRegressionDelta` captures the differences between two `Slices` as additions represented by `AddSliceElement` and removals denoted by `RemoveSliceElement`. Both,

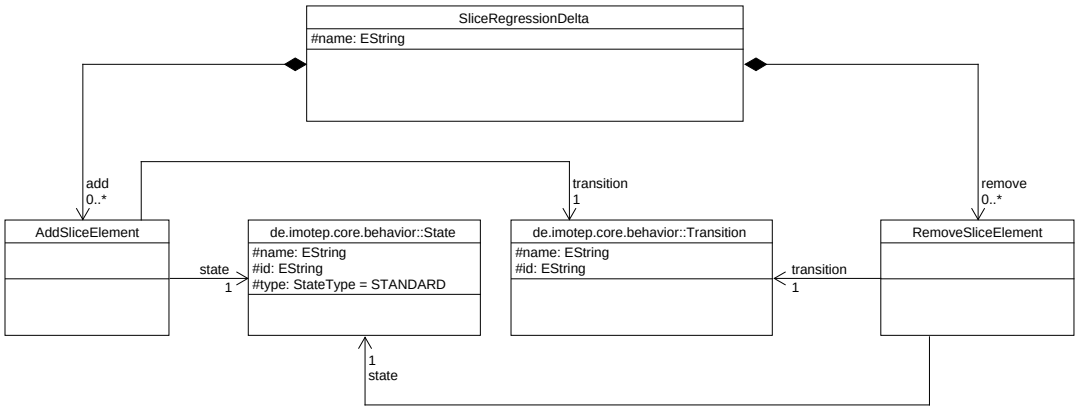


Figure 4.15: Meta Model of the Incremental Model Slicing Plug-In (Delta-Oriented Part)

`AddSliceElement` as well as `RemoveSliceElement` are mapped to the state machine elements `State` and `Transition`, respectively. A `SliceRegressionDelta` indicates the result of the change impact analysis w.r.t. a `SlicingCriterion` and is, therefore, exploited in our model-based SPL regression testing framework and its prototypical implementation (cf. Chapt. 5).

Higher-Order Delta Application Reasoning. The plug-in `de.imotep.dope.dependency` realizes the reasoning about the application of a `HigherOrderDelta` to facilitate change impact analysis between two consecutive SPL versions under test as described in Sect. 4.2. For a better illustration, we split the meta model in four parts. In Fig. 4.16, the main part of the meta model is depicted, whereas in Fig. 4.17 the relation between delta nodes and delta dependency edges, in Fig. 4.18 the delta-oriented part, and in Fig. 4.19 the variant tree part is shown. The main class of the meta model depicted in Fig. 4.16 is `DeltaDependencyGraph`. A `DeltaDependencyGraph` captures the sets of `DeltaNodes` and `DeltaDependencyEdges`. Each `DeltaNode` is mapped to the `Delta` it represents. A `DeltaDependencyEdge` has a `DeltaNode` as `sourceNode` and is either a `SingleTargetDependencyEdge` or a `MultiTargetDependencyEdge`. The classes inherit from `DeltaDependencyEdge` to realize the delta dependency types as defined in Sect. 4.2. The single-target dependencies are realized by the classes `CompleteOptionalDependencyEdge`, `PartialOptionalDependencyEdge`, `MandatoryDependencyEdge`, and `ExclusiceDependencyEdge`. For those classes, the `targetNode` is referenced from their superclass `SingleTargetDependencyEdge`. In contrast, the multi-target dependencies are implemented via `NonExclusiveAlternativeDependencyEdge`, `CompleteAlternativeDependencyEdge`, `ImplicationDependencyEdge`, `OptionalAlternativeDependencyEdge`, `NonExclusivePartialAlternativeDependencyEdge`, `PartialOptionalAlternativeDependencyEdge`, `CompleteOptionalAlternativeDependencyEdge`, and `PartialAlternativeDependencyEdge`. The set of `targetNodes` is referenced from their superclass `MultiTargetDependencyEdge`. As depicted in Fig. 4.17, each `DeltaNode` is mapped to all incoming and outgoing dependencies it is involved in.

A `DeltaDependencyGraph` is created for a set of `Deltas` and a corresponding `FeatureModel` by executing the method `buildDependencyGraph()`. The incremental adaptation is handled via the method `incrementalDeltaDependencyGraphAdaption()`. To capture the differences, the class `DeltaDependencyGraphRegressionDelta` is defined. As depicted in Fig. 4.18, a `DeltaDependency-`

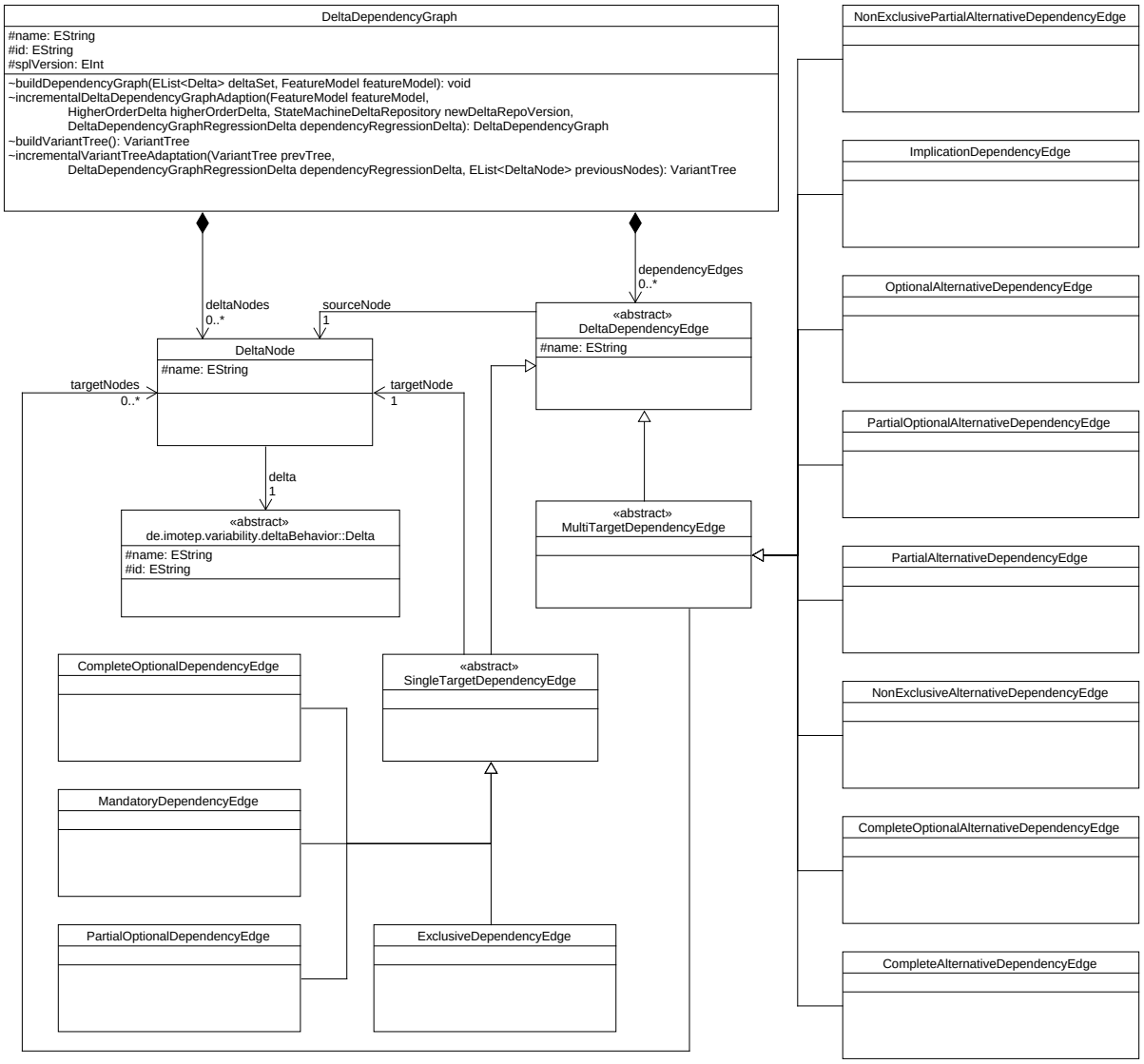


Figure 4.16: Meta Model of the Higher-Order Delta Application Reasoning Plug-In (Main Part)

`GraphRegressionDelta` is mapped to the `DeltaDependencyGraph` it transforms and the result via the references `fromGraph` and `toGraph`, respectively. In addition, each `DeltaDependencyGraphRegressionDelta` captures the additions and removals of `DeltaNodes` and `DeltaDependencyEdges` and also the modifications of `DeltaNodes`.

After the delta dependency analysis has finished, a `VariantTree` can be derived by executing the method `buildVariantTree()` of `DeltaDependencyGraph`. As shown in Fig. 4.19, a `VariantTree` is realized as composition, i.e., a `VariantTree` has a reference to its parent and can have a left child `VariantTree` and a right child `VariantTree` to establish the hierarchy, and is captured by the `DeltaDependencyGraph`. Based on the functions `isLeaf()` and `isRoot()`, we are able to determine if a `VariantTree` is a leaf or a root tree node, respectively. Furthermore, each `VariantTree` is mapped to its representing `DeltaNode` and to a set of `DeltaDependencyEdges` denoting its restrictions. The set of all variant-specific delta sets is represented as a set of `VariantTreePaths`

Figure 4.17: Meta Model of the Higher-Order Delta Application Reasoning Plug-In (Dependency Part)

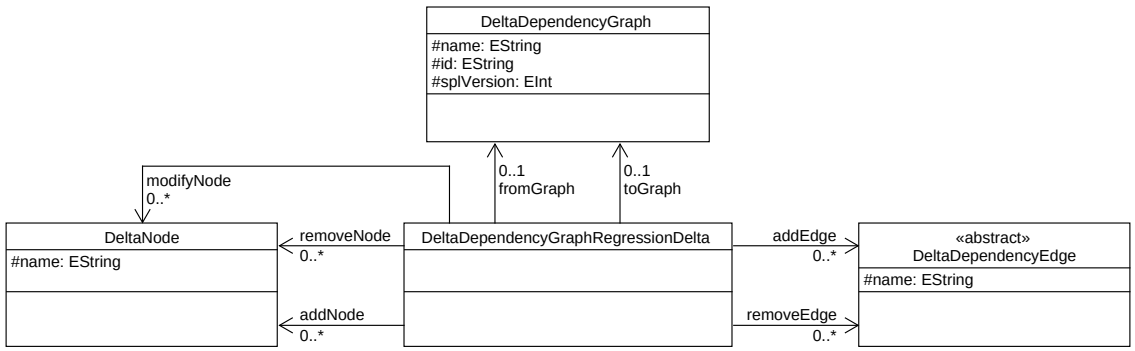


Figure 4.18: Meta Model of the Higher-Order Delta Application Reasoning Plug-In (Delta-Oriented Part)

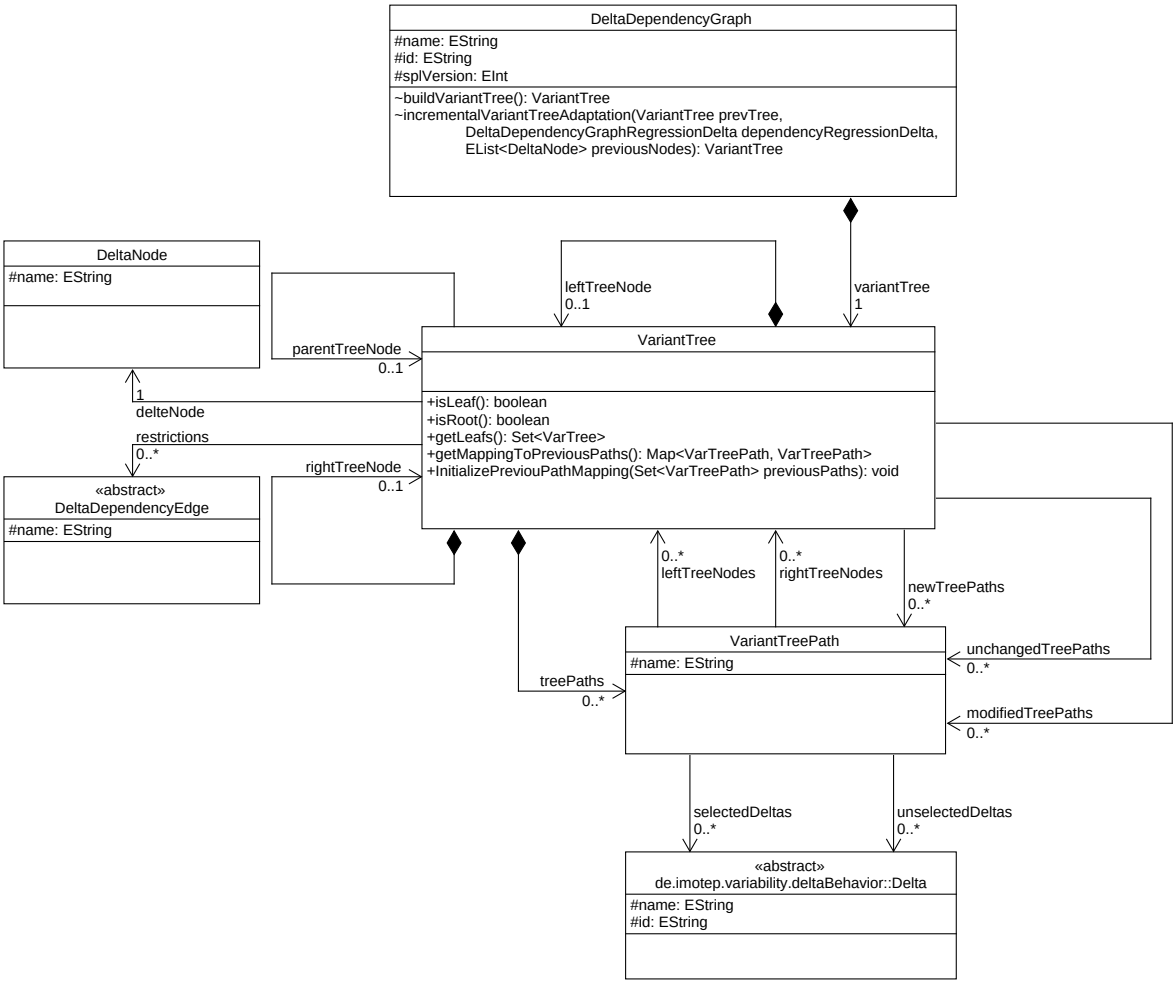


Figure 4.19: Meta Model of the Higher-Order Delta Application Reasoning Plug-In (Variant Tree Part)

captured by a `VariantTree`. We obtain the set of all delta sets by accessing the captured set of `VariantTreePaths` via the reference `treePaths`. A `VariantTree` is incrementally computed via the method `incrementalVariantTreeAdaptation()` of `DeltaDependencyGraph`, where also the categorization of `VariantTreePaths` is performed. We obtain the categorized sets of delta sets by accessing the respective references of `unchangedTreePaths`, `modifiedTreePaths`, and `newTreePaths`. For the `VariantTreePaths` categorized as `unchangedTreePaths` or `modifiedTreePaths`, we get the mapping to the previous `VariantTreePaths` via the function `getMappingToPreviousPaths()`.

Based on those plug-ins, we are able to apply the change impact analysis techniques for evolving delta-oriented SPLs under test. The results are exploited in our model-based SPL regression testing framework to guide the retest test selection (cf. Chapt. 5). The plug-ins are provided online as part of the prototype of the regression testing framework.[4] In the following section, we present the evaluation of our change impact analysis techniques and their prototypical implementations.

4.3.2 Evaluation of Change Impact Analyses

In this section, we present the evaluation of our change impact analysis techniques to validate their applicability and efficiency. First, we formulate the research questions and describe the methodology of the evaluation. Second, we present and discuss our obtained results and the threats to the validity of our evaluation.

Research Questions and Methodology

The evaluation of our delta-oriented change impact analysis techniques is defined as *controlled experiment*, where we apply the incremental model slicing approach as well as the reasoning process and their prototypical tool support to the three evolving subject SPLs (cf. Sect. 3.4.2). For the documentation of the research methodology, we followed the guidelines defined by Wohlin et al. [WHH03; WRH+12] as well as Juristo and Moreno [JM13]. To conduct the experiment, we formulate the following research questions (RQ) to be answered.

RQ1 Is incremental model slicing *applicable as change impact analysis of variants* by identifying differences in the execution dependencies between slices of variant-specific state machine test models computed for the same slicing criterion?

RQ2 Is incremental model slicing a *more efficient change impact analysis technique* in terms of required analysis time compared to the application of standard model slicing?

- **a.** Do we achieve a *decrease in the time required for dependency analysis and dependency graph generation* based on the incremental dependency graph adaptation compared to the standard dependency graph generation?
- **b.** Do we achieve a *decrease in the time required for the slice computation and the slice difference derivation* based on the incremental slice recomputation compared to the standard slice computation?

RQ3 Is the reasoning about higher-order delta application *applicable as change impact analysis of versions of variants* by identifying the changes to the variant set in terms of unchanged, new, and modified variants?

RQ4 Is the reasoning about higher-order delta application a *more efficient change impact analysis* in terms of required analysis time compared to the product-by-product impact analysis?

[4] `https://github.com/SLity/mbtSPLregression`

To answer the defined research questions, we determine both *qualitative* as well as *quantitative* data. The higher-order delta test models of the three subject systems and also the analysis artifacts, i.e., dependency graphs, slices, variant trees etc., created during the execution of the experiment define the qualitative data. We execute the prototypical implementations on a machine with 32 Intel Xenon E5 (3.1GHz) cores and 50GB RAM running Ubuntu 18.04.1 LTS x86_64 as operating system. By recording the runtime of the analysis techniques and also by applying metrics, e.g., number of slice differences, to the computed analysis artifacts, we obtain quantitative data. We exploit the quantitative data to facilitate a *hypothesis confirmation*, where we use the defined research questions as hypothesis. To investigate whether a hypothesis can be confirmed, we apply the following data analysis and research methodology.

For the evaluation of our incremental model slicing technique, we aim to answer the research questions **RQ1** to reason about its applicability as change impact analysis technique, and **RQ2** to show that it is efficiently applicable for change impact analysis. As the incremental model slicing technique is divided into the subprocesses of incremental dependency graph adaptation as well as incremental slice computation, we divide **RQ2** into two respective sub-research questions, accordingly. To answer **RQ1**, we examine whether our incremental model slicing technique detects differences between slices of consecutively tested variants computed for the same slicing criterion if they exist. Slice differences indicate the impact on changes applied and, therefore, indicate retest potentials to be handled by our regression testing framework (cf. Chapt. 5). As slicing criteria, we focus on transitions as they are also used as test goals in our testing framework to guide, e.g., the test-case generation. In addition, we apply standard model slicing with a subsequent slice difference derivation as a *baseline* to ensure that we obtain correct results. To answer **RQ2** as well as its sub-questions **RQ2a** and **RQ2b**, we measure the time required for the generation of the dependency graph and the slice computation including the derivation of slice differences w.r.t. the execution of both techniques, i.e., incremental model slicing and standard slicing, for a comparison.

We obtain the respective quantitative data for both research questions, e.g., by means of the number of slices with as well as without differences and the computation time, by executing the incremental model slicing and standard model slicing techniques for each subject SPL and its versions. For one SPL version θ, we step from each variant $v_i \in \mathbb{V}_\theta$ to each other variant $v_j \in \mathbb{V}_\theta$ and apply the slicing techniques, where we compute the respective state machine regression deltas to be used as input for the incremental slicing computation. We repeat such complete executions for each subject SPL 100 times to provide reliable results in order to answer research question **RQ2**. The repetitions also establish a reliability against random outliers of the execution time occuring due to delays, e.g., based on the Java garbage collection. Furthermore, to investigate whether our slicing technique has a better performance by means of execution time compared to standard slicing with subsequent slice difference determination, we perform a hypothesis test [WRH+12] using the tool R[5] for statistical computing. The data, i.e., the measured execution times, used as random samples is non-parametric as we have no information whether the runtime distribution follows a standard distribution. We also cannot assume that the results of the standard slicing and our technique have the same variance. In addition, our data is not paired. Based on those prerequisites, we apply the Wilcoxon-Mann-Whitney-Test [WRH+12] to validate the hypothesis that our technique has a better performance, where we further focus on a one-sided test. A two-sided test would validate whether

[5] https://www.r-project.org/, last access: May 31st, 2019

the median μ of the execution times of both techniques differ, whereas the one-sided test facilitates the validation that the standard technique requires more time than our slicing technique. Therefore, we define the following null and alternative hypothesis:

$$H_0 : \mu_{Std} \leq \mu_{Incr} \text{ and } H_1 : \mu_{Std} > \mu_{Incr}$$

We apply the hypothesis test separately for the dependency analysis and the slice computation performed by our slicing technique and standard slicing, where we use 5% as significance level. The execution of the hypothesis tests result in p-values [WRH+12]. In case the value of $p < 0,05$ is lower than the significance level, we confirm the alternative hypothesis and reject the null hypothesis.

For the evaluation of the reasoning about the higher-order delta application, we answer the research questions **RQ3** to investigate its applicability as change impact analysis between consecutively tested SPL versions, and **RQ4** to show its efficiency in terms of the required analysis times. To answer **RQ3**, we determine the categorization of changes to the variant set by means of unchanged, modified, and new variants when stepping from one SPL version θ_i to the next version θ_{i+1} following the sequential evolution history of the subject SPLs. We ensure the correctness of the categorization by comparing the results to the original definition and manual impact classification of the evolving delta-oriented SPLs [NLS18]. To answer **RQ4**, we record the time required for the generation of the delta dependency graph and the variant tree computation including the derivation and categorization of variant-specific delta sets w.r.t. the execution of our reasoning technique. In contrast to the slicing evaluation, where we compare the analysis times of our incremental technique to those of the standard slicing, we leave out a comparison of the analysis times of our reasoning approach to those of the naive product-by-product approach (cf. Sect. 4.2). This is justified by two reasons.

First, the automated detection of modified and removed variants is not straightforward or even not possible without the interaction with a domain expert, e.g., test engineer. For instance, by comparing all state machines of both SPL versions in a brute force way, we would determine unchanged variants. State machines of the previous SPL version which do not have a direct partner can be categorized as removed and state machines of the new SPL versions which do not have a partner can be categorized as new. However, such simple categorization does not allow for the identification of modified state machines. For the identification of modifications, we have to determine how state machine variants differ between SPL versions. A naive approach would be to categorize the state machines which are the most similar to each other as modified, where again a categorization may become problematic if there are more than one partner w.r.t. equal numbers of distinct differences. Hence, the naive comparison and categorization is inadequate and requires the interaction with a domain expert to identify the correct pairs of state machines for the modified category.

Second, the time required for a product-by-product comparison of variant-specific state machines of consecutive SPL versions mainly depends on the size of the state machines, whereas our reasoning technique depends on the number of state machine deltas. Hence, with an increasing size of variant-specific state machines to be compared, the analysis time will also increase. For our reasoning approach, the number of deltas to be analyzed predefine the number of satisfiability checks to derive the delta dependencies. Those checks performed by executing SAT solver like SAT4j[6] require the most time of the delta dependency graph generation. Since SAT solvers are improved by

[6] http://www.sat4j.org/, last access: May 31st, 2019

new heuristics to solve a given satisfiability problem, it is likely that for the same input our reasoning will get rather faster in the future. Similar to **RQ2**, we repeat the executions for each subject SPL 100 times to provide reliable results in order to answer research question **RQ4**.

Furthermore, for the efficiency evaluation of our variant-set change impact analysis, we focus solely on the analysis runtimes and abstract from a distinct evaluation of the size of variant trees by means of number of tree nodes. According to Leung and White [LW91] and their defined cost model for regression testing, the time required for the change impact analysis is a crucial cost factor for a regression testing technique to be efficient. Obviously, the size of a variant tree has an influence on the runtime of the change impact analysis as the creation as well as its incremental adaptation depend on the number of tree nodes to be analyzed. Therefore, the size of variant trees is implicitly evaluated in the context of the runtime analysis.

Results

We present and discuss the results of our evaluation individually for the defined research questions.

RQ1. In Tab. 4.6, we summarize the results of the application of our incremental model slicing to the three evolving delta-oriented subject SPLs (cf. Sect. 3.4). In the second column, we provide the total number of slices computed during the execution. We present the data of solely one complete execution out of the 100 repetitions as all of them provide the same results regarding the number and computation of slices and slice differences. The total number of slices computed depends on the number of variants of an SPL version and the size of the variant-specific state machines in terms of number of contained transitions as we compute for each transition used as slicing criterion a respective slice. The third column captures the number of slices which are incrementally computed. The fourth column contains the number of incrementally computed slices, where slice differences are detected. In the fith column, we provide the number of incrementally computed slices, where no slice differences are detected. The sixth column captures the maximal amount of slice differences, whereas the seventh column comprises the minimal amount of slice differences. In the last column, the average amount of slice differences w.r.t. all incrementally computed slices is provided.

For all subject systems and their versions, we have a higher number of computed slices compared to the number of incrementally computed slices. The application of our slicing technique to incrementally compute a slice for a given slicing criterion requires that in both variants which are subsequently analyzed the same slicing criteria exists. Hence, we solely apply our incremental technique for the common parts of consecutive variants to be analyzed and apply standard model slicing for the variable parts those variants differ in. That means, we cannot compute all slices of a consecutive analyzed variant incrementally resulting in the lower number. In addition, the presented numbers provide solely information about the incremental recomputation of slices. In contrast, the incremental dependency graph adaptation is always applicable. For the evolving SPLs, where we step from each variant $v_i \in \mathbb{V}_\theta$ to each other variant $v_j \in \mathbb{V}_\theta$ of a respective version, we compute on average 50% of the slices incrementally (cf. Tab. 4.6). However, when we apply our slicing technique as change impact analysis in our model-based regression testing framework, a given testing order of variants may have an impact on those percentages as the size of common as well as variable parts between subsequently tested variants differ w.r.t. different testing orders. We discuss and evaluate the impact of testing orders to the application of incremental model slicing as change impact analysis in Chapt. 5.

Table 4.6: Results of the Application of Incremental Model Slicing for the Evolving SPLs **W**iper, **V**ending **M**achine, and **M**ine **P**ump (# = Number, ⌀ = Average)

SPL	#Slices	#Incremental (%)	$\lvert\Delta^{slice_c}_{v_i^\theta,v_j^\theta}\rvert > 0$	$\lvert\Delta^{slice_c}_{v_i^\theta,v_j^\theta}\rvert = 0$	Max $\lvert\Delta^{slice_c}_{v_i^\theta,v_j^\theta}\rvert$	Min $\lvert\Delta^{slice_c}_{v_i^\theta,v_j^\theta}\rvert$	⌀ $\lvert\Delta^{slice_c}_{v_i^\theta,v_j^\theta}\rvert$
W_{θ_0}	347	302 (87.03)	212	90	15	2	8.16
W_{θ_1}	610	358 (58.68)	214	144	26	2	15.65
W_{θ_2}	1,802	941 (52.21)	557	384	31	2	16.80
W_{θ_3}	8,692	4,520 (52.00)	2,824	1,696	43	2	20.11
W_{θ_4}	9,466	4,718 (49.84)	3,022	1,696	43	2	20.04
VM_{θ_0}	4,422	3,454 (78.10)	3,454	0	15	2	6.99
VM_{θ_1}	15,214	7,925 (52.09)	7,925	0	30	2	13.73
VM_{θ_2}	43,206	23,905 (55.32)	23,905	0	30	2	14.32
VM_{θ_3}	13,172	7,631 (57.93)	7,631	0	23	2	11.50
VM_{θ_4}	19,056	9,123 (47.87)	8,571	552	35	2	11.50
VM_{θ_5}	20,904	9,675 (46.28)	8,571	1,104	35	2	11.50
VM_{θ_6}	30,257	17,329 (57.27)	13,885	3,444	41	2	13.50
MP_{θ_0}	3,048	2,456 (80.57)	1,316	1,140	42	1	8.08
MP_{θ_1}	4,144	2,568 (61.96)	1,316	1,252	28	1	6.26
MP_{θ_2}	18,260	11,604 (63.54)	5,812	5,792	38	1	6.46

To answer **RQ1**, we examine whether our technique detects slice differences. As we can see in Tab. 4.6, our incremental slicing technique computes slices with ($\lvert\Delta^{slice_c}_{v_i^\theta,v_j^\theta}\rvert > 0$) and without ($\lvert\Delta^{slice_c}_{v_i^\theta,v_j^\theta}\rvert = 0$) slice differences. Hence, we are able to detect changing execution dependencies in the shared common behavior between subsequently analyzed variants if such changes exist. Detected changes of the execution dependencies represent the impact of changes between the respective variant-specific state machine test models. By comparing both numbers, we also see that the number of slices with differences is higher than the number of slices without differences. That means, for the selected subject SPLs, the changes between subsequently analyzed variants have an impact on their common behavior. For the SPL versions θ_0 to θ_3 of the Vending Machine SPL, we identified special cases, where we always compute and determine slices with differences. In those cases, the changes always have an impact on shared common behavior as the functionality is specified in a single state machine region. Starting with version θ_4, the functionality is specified in different regions and, therefore, we are able to compute slices without differences.

By examining the maximal, minimal, and average number of determined slice differences, we see that changes between subsequently analyzed variants have a varying impact. For the Wiper as well as Vending Machine SPL and their versions, the minimal number of slice differences is two, whereas for the evolving Mine Pump SPL the minimal number is one. That means, in some cases, the impact of changes and, therefore, the identified retest potentials are small. In contrast, the maximal number which also depends on the size of the variant-specific state machines to be sliced ranges (1) from 15 to 43 for the Wiper SPL, (2) from 15 to 41 for the Vending Machine system, and (3) from 42 to 38 for the Mine Pump SPL. On average, the number of slice differences w.r.t. the number of incrementally computed slices is more or less half of the maximal number of slice differences.

In summary, the results of the application of incremental model slicing to the three evolving subject SPLs show its applicability as change impact analysis technique (**RQ1**). By incorporating

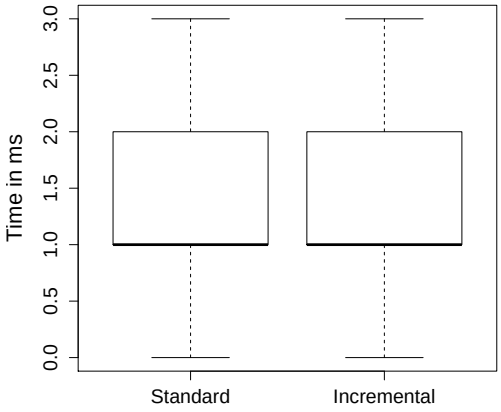
the changes of the state machines of subsequently analyzed variants, we are able to detect the impact to shared common behavior as slice differences. Slice differences represent changes of the execution dependencies of the respective slicing criterion indicating retest potentials to be handled by our regression testing framework (cf. Chapt. 5). Based on the results, we further infer that our incremental slicing technique will also detect changing execution dependencies when applied between a variant and its modified version during regression testing of consecutive SPL versions.

RQ2. We investigate the efficiency of our incremental model slicing technique compared to standard model slicing including the slice difference derivation used as a baseline (**RQ2**) by assessing the execution times when applied for the three SPLs and their versions. As our slicing technique is divided into the subprocesses of incremental dependency graph adaptation as well as incremental slice computation, we provide the runtime results also divided for both subprocesses. The results presented and discussed in the following are obtained based on the 100 repetitions of a complete execution for each SPL version. As defined in the methodology description, in a complete execution, we step from each variant $v_i \in \mathbb{V}_\theta$ to each other variant $v_j \in \mathbb{V}_\theta$ and apply both slicing techniques.

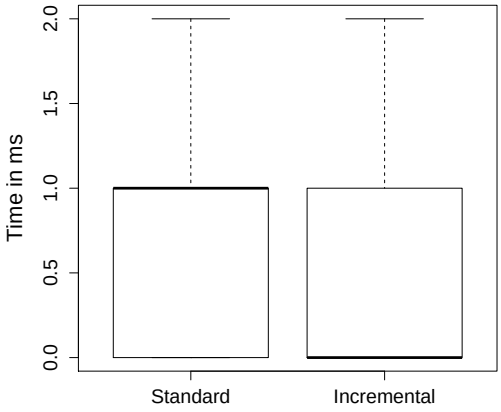
In Fig. 4.20, the runtime results for the dependency analysis and slicing computation for the versions θ_0, θ_2, and θ_4 of the Wiper SPL are depicted in respective box plots. We focus on those versions as the box plots of the remaining versions θ_1 and θ_3 of the Wiper SPL show very similar runtime distributions and, hence, provide no further gain in information. The omitted box plots can be found in the appendix A in Fig. A.1. For the initial Wiper SPL version θ_0, we see that the runtimes of the dependency analysis depicted in Fig. 4.20a are equal sharing the same median of one milliseconds. This is due to the rather small size of variant-specific state machines, whereas our incremental technique requires some extra time to derive the dependency graph regression delta in contrast to the standard dependency analysis. For the subsequent versions (cf. Fig. 4.20c and Fig. 4.20e), where based on newly introduced functionality the size of state machines increases (cf. Tab. 3.4), the runtimes for the dependency analysis of our technique are better compared to the standard technique. There are some cases, where both techniques require 0 ms. Those runtimes are obtained when analyzing the small variants of the Wiper SPL existing in all SPL versions. We have similar results for the slice computation. In the initial SPL version θ_0, the runtimes are quite equal, whereas our slicing technique has a slightly better performance for the remaining Wiper SPL versions compared to the standard technique. However, both techniques are very fast as they require solely some milliseconds (max 7 ms) for the slice computation and slice difference derivation.

We see a similar tendency of runtime distributions as for the Wiper SPL in the results for the Vending Machine shown in Fig. 4.21 and Mine Pump SPL depicted in Fig. 4.22. In the initial Vending Machine SPL version θ_0, the runtimes of the dependency analysis and slice computation shown in Fig. 4.21a and Fig. 4.21b are quite equal. The same holds for the initial version of the Mine Pump SPL (cf. Fig. 4.22a and Fig. 4.22b). With proceeding versions, our technique has a better performance in both subprocesses for both evolving subject SPLs. Again, this is due to the increasing number of variants and size of variant-specific state machines resulting from newly introduced functionality. Similar as for the Wiper SPL, we omitted some SPL versions and their runtime results represented as box plots for the Vending Machine. We refer the reader to the appendix A, where in Fig. A.2 and in Fig. A.3 the box plots for the Vending Machine SPL versions θ_1, θ_2, θ_3, and θ_5 are shown.

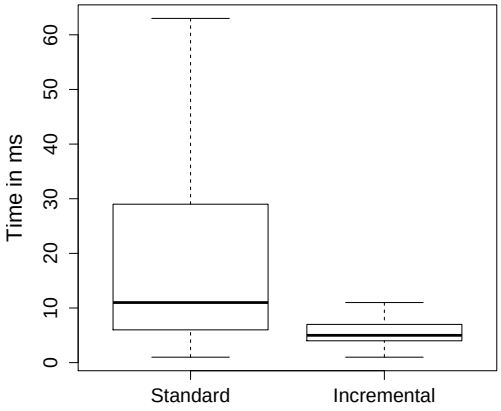
As described above, we performed 100 repetitions of a complete execution for each SPL version of each subject system to provide results which are reliable against runtime outliers. The percent-



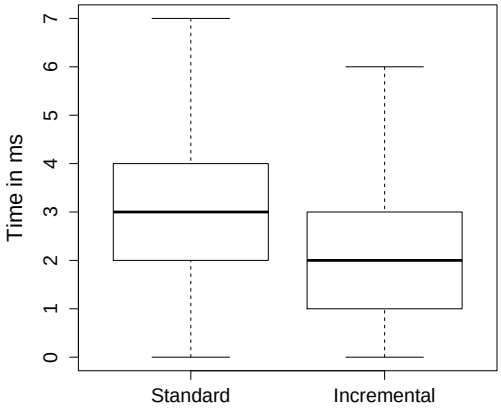
(a) Dependency Analysis Runtime for Wiper SPL Version θ_0



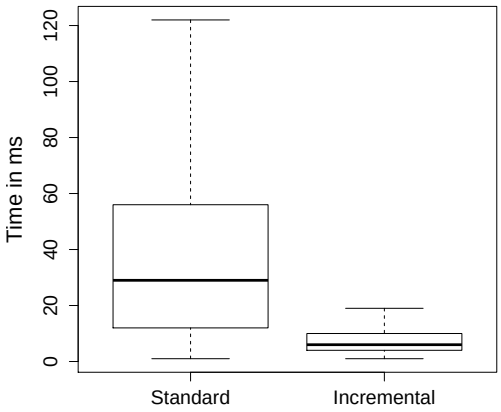
(b) Slicing Runtime for Wiper SPL Version θ_0



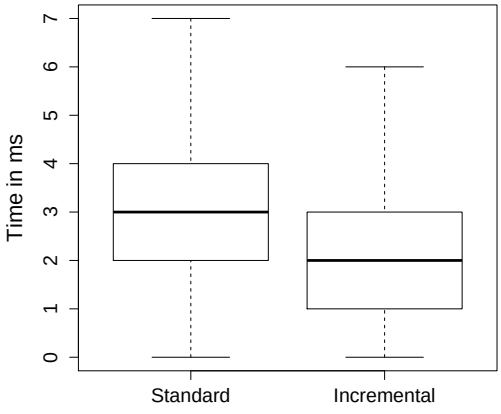
(c) Dependency Analysis Runtime for Wiper SPL Version θ_2



(d) Slicing Runtime for Wiper SPL Version θ_2

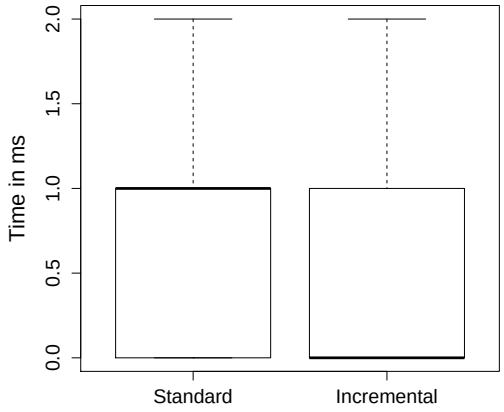


(e) Dependency Analysis Runtime for Wiper SPL Version θ_4

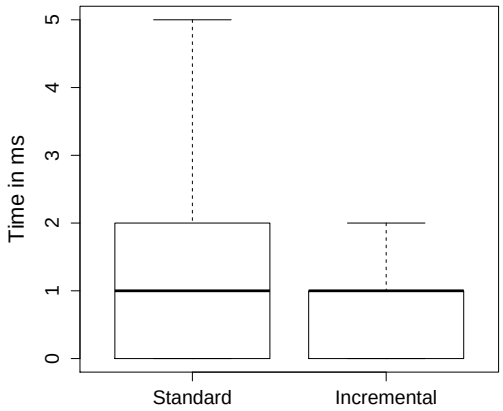


(f) Slicing Runtime for Wiper SPL Version θ_4

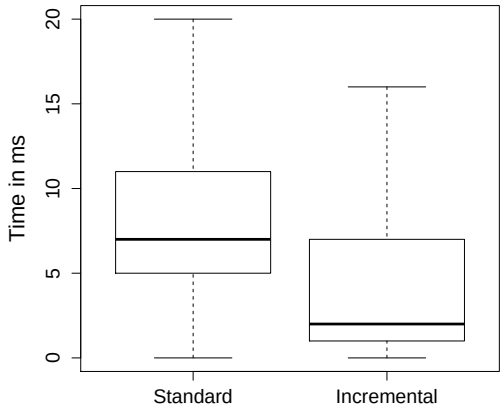
Figure 4.20: Runtime Results of the Dependency Analysis and Slice Computation for the Versions θ_0, θ_2, and θ_4 of the Wiper SPL



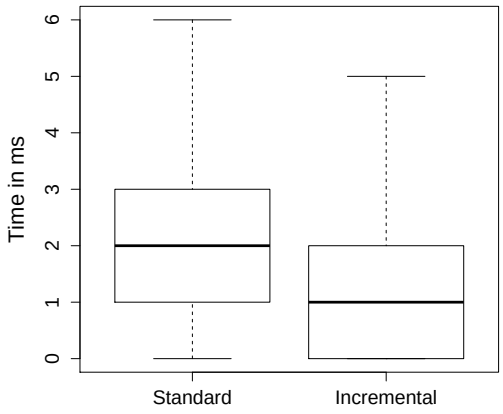
(a) Dependency Analysis Runtime for Vending Machine SPL Version θ_0



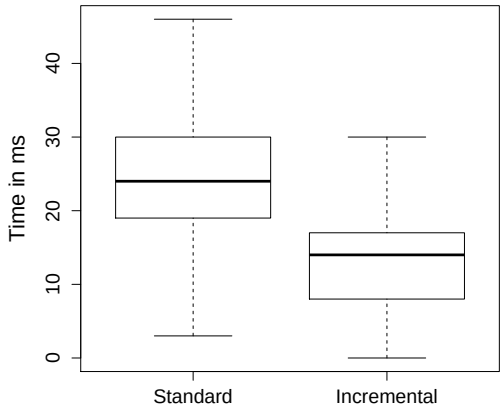
(b) Slicing Runtime for Vending Machine SPL Version θ_0



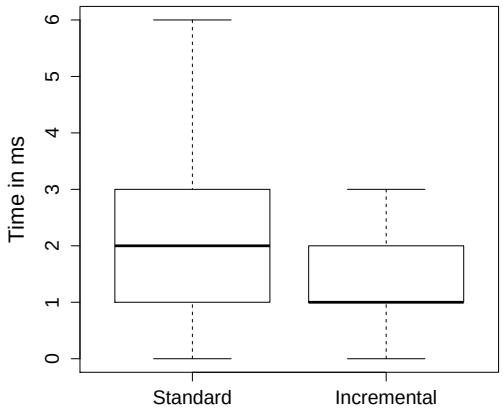
(c) Dependency Analysis Runtime for Vending Machine SPL Version θ_4



(d) Slicing Runtime for Vending Machine SPL Version θ_4

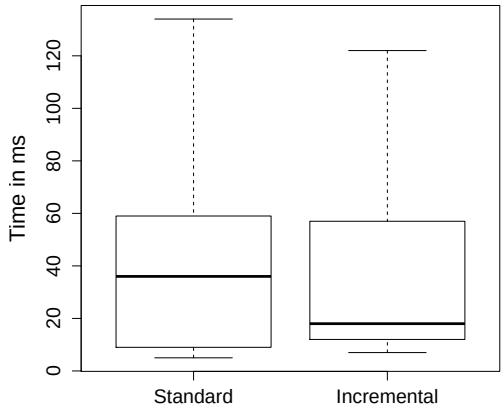


(e) Dependency Analysis Runtime for Vending Machine SPL Version θ_6

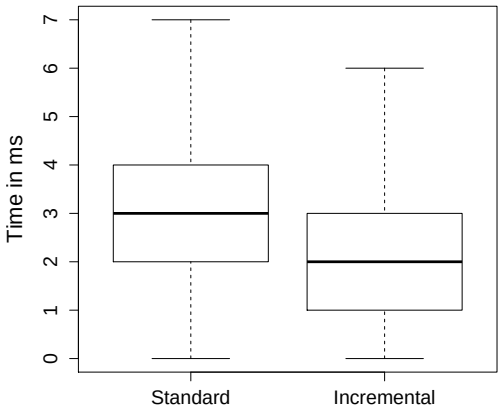


(f) Slicing Runtime for Vending Machine SPL Version θ_6

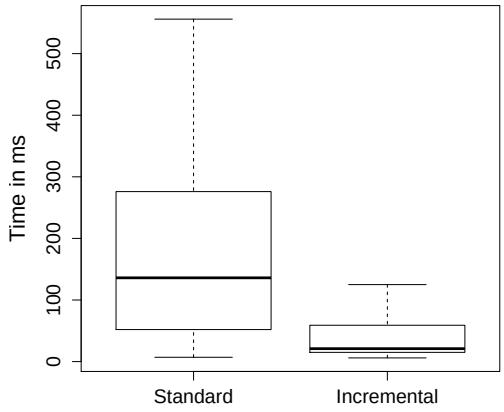
Figure 4.21: Runtime Results of the Dependency Analysis and Slice Computation for the Versions θ_0, θ_4, and θ_6 of the Vending Machine SPL



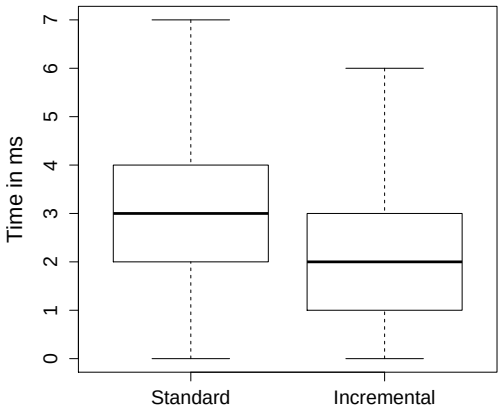
(a) Dependency Analysis Runtime for Mine Pump SPL Version θ_0



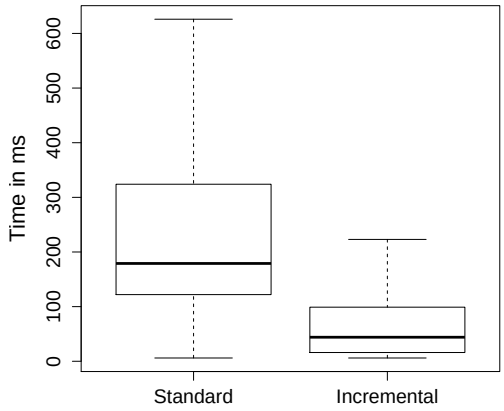
(b) Slicing Runtime for Mine Pump SPL Version θ_0



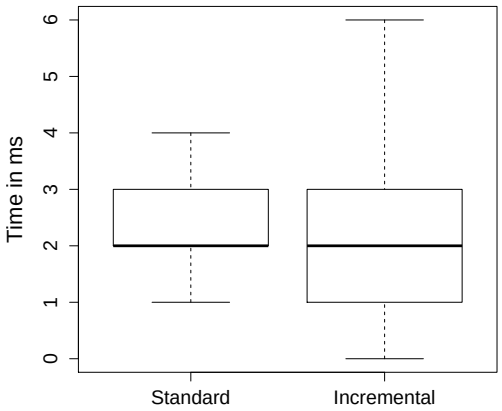
(c) Dependency Analysis Runtime for Mine Pump SPL Version θ_1



(d) Slicing Runtime for Mine Pump SPL Version θ_1



(e) Dependency Analysis Runtime for Mine Pump SPL Version θ_2



(f) Slicing Runtime for Mine Pump SPL Version θ_2

Figure 4.22: Runtime Results of the Dependency Analysis and Slice Computation for the Versions θ_0, θ_1, and θ_2 of the Mine Pump SPL

ages of outliers w.r.t. the total number of executions for each evolving SPL is summarized in the appendix A in Tab. A.1. On average, 5% of the executions are classified as outliers. By examining the outliers, we did not find any reasons for their occurrences as there were no repetitions for certain variant combinations allowing for a deduction of such reasons. We classify those runtime outliers to be random occurring due to delays, e.g., based on the execution of the Java garbage collection or operating system interruptions. Furthermore, we did not incorporate and, therefore, abstract from the outliers in the box plots of the three subject SPLs and their versions as (1) they occur randomly and (2) the number of outliers represented by the percentages are rather small.

For the one-sided Wilcoxon-Mann-Whitney-Test [WRH+12], we summarized all resulting p-values in the appendix A in Tab. A.2. The results of the one-sided test reveal that for the slice computation, our technique has a better performance than the application of standard slicing, where we always determine a p-value $p = 2.2e^{-16} < 0.05$ lower than the significance level of 5%. In the context of the dependency analysis, this is not always the case. For the initial version θ_0 of the Wiper and Mine Pump SPL as well as for the versions θ_1 and θ_3 of the Vending Machine SPL, the p-value $p = 1$ equals one such that we cannot reject the null hypothesis. However, as depicted in the boxplots of those cases in Fig. 4.20 for the Wiper SPL, in Fig. A.2 for the Vending Machine SPL, and in Fig. 4.22 for the Mine Pump SPL, we see that the runtimes of the dependency analysis performed by our technique and the standard technique are very similar. We assume that the four cases can be neglected for the overall result as for the other versions of the three subject SPLs, where the state machine sizes increase in addition, our technique achieves a better performance than the standard technique.

Table 4.7: Results of the Runtime Comparison for the Dependency Analysis between Standard Slicing (Std) and Incremental Slicing (Incr) for the **W**iper, **V**ending **M**achine, and **M**ine **P**ump SPL (# = Number, ↑ = Faster Times, ⌀ = Average, ∂ = Time in Milliseconds, ↔ = Equal Times)

SPL	#Executions	Dependency Analysis				
		#Std↑ (%)	$\varnothing\partial(Std) - \partial(Incr)$	#Incr↑ (%)	$\varnothing\partial(Incr) - \partial(Std)$	#Std↔Incr (%)
W_{θ_0}	2,800	1,062 (37.92)	1.22	361 (12.89)	1.26	1,377 (49.17)
W_{θ_1}	2,800	362 (12.92)	1.07	1,971 (70.39)	2.52	467 (16.67)
W_{θ_2}	6,600	1,211 (18.34)	2.48	4,970 (75.30)	13.77	419 (6.34)
W_{θ_3}	27,600	2,620 (9.49)	2.56	22,160 (80.28)	18.47	2,820 (10.21)
W_{θ_4}	27,600	1,086 (3.93)	3.82	25,807 (93.50)	36.58	707 (2.56)
VM_{θ_0}	37,800	11,582 (30.64)	1.08	15,548 (38.48)	1.08	11,670 (30.87)
VM_{θ_1}	86,100	28,706 (33.34)	1.27	23,626 (27.44)	1.06	33,768 (39.21)
VM_{θ_2}	241,500	73,240 (30.32)	1.14	74,895 (31.01)	1.06	93,365 (38.66)
VM_{θ_3}	86,100	30,339 (35.23)	1.18	24,034 (27.91)	1.06	31,727 (36.84)
VM_{θ_4}	86,100	17,440 (20.25)	2.52	62,145 (72.17)	5.73	6,515 (7.56)
VM_{θ_5}	86,100	11,937 (13.86)	2.93	67,521 (78.42)	16.21	6,642 (7.71)
VM_{θ_6}	112,800	11,747 (10.41)	5.29	96,756 (85.77)	16.05	4,297 (3.80)
MP_{θ_0}	12,000	7,069 (58.90)	7.47	4,777 (39.80)	47.20	154 (1.28)
MP_{θ_1}	12,000	811 (6.75)	11.25	11,121 (92.67)	121.23	68 (0.56)
MP_{θ_2}	49,600	1,880 (3.79)	14.26	47,626 (96.02)	150.42	94 (0.18)

In addition to the results presented so far, we also compared the runtimes on a more detailed level to determine w.r.t. the total number of executions (1) how often the standard technique is faster, (2) how often our technique is faster, and (3) how often the runtimes of both techniques

Table 4.8: Results of the Runtime Comparison for the Slice Computation between Standard Slicing (Std) and Incremental Slicing (Incr) for the **W**iper, **V**ending **M**achine, and **M**ine **P**ump SPL (# = Number, ↑ = Faster Times, ⌀ = Average, ∂ = Time in Milliseconds, ↔ = Equal Times)

SPL	#Executions	Slicing				
		#Std↑ (%)	$\varnothing\partial(Std) - \partial(Incr)$	#Incr↑ (%)	$\varnothing\partial(Incr) - \partial(Std)$	#Std↔Incr (%)
W_{θ_0}	2,800	640 (22.85)	1.13	1,480 (52.85)	1.26	680 (24.28)
W_{θ_1}	2,800	514 (18.35)	1.28	1,685 (60.17)	1.66	601 (21.46)
W_{θ_2}	6,600	1,617 (24.50)	1.65	3,762 (57.00)	2.16	1,221 (18.50)
W_{θ_3}	27,600	4,875 (17.66)	1.64	17,878 (64.77)	22.34	4,847 (17.56)
W_{θ_4}	27,600	6,286 (22.77)	1.98	16,083 (58.27)	2.26	5,231 (18.95)
VM_{θ_0}	37,800	10,244 (27.10)	1.33	18,462 (48.84)	1.45	9,094 (24.05)
VM_{θ_1}	86,100	16,667 (19.35)	1.30	51,155 (59.41)	1.63	18,278 (21.22)
VM_{θ_2}	241,500	48,307 (20.00)	1.28	139,253 (57.66)	1.50	53,940 (22.33)
VM_{θ_3}	86,100	17,988 (20.89)	1.29	49,091 (57.01)	1.47	19,021 (22.09)
VM_{θ_4}	86,100	17,985 (20.88)	1.46	50,770 (58.96)	1.80	17,345 (20.14)
VM_{θ_5}	86,100	19,138 (22.22)	1.47	48,480 (56.30)	1.66	18,482 (21.46)
VM_{θ_6}	112,800	25,950 (23.00)	1.55	62,265 (55.19)	1.85	24,585 (21.79)
MP_{θ_0}	12,000	2,711 (22.59)	1.71	7,062 (58.85)	2.40	2,227 (18.55)
MP_{θ_1}	12,000	3,418 (28.48)	2.06	6,642 (55.35)	2.49	1,940 (16.16)
MP_{θ_2}	49,600	14,978 (30.19)	1.89	24,411 (49.21)	2.17	10,211 (20.58)

are equal. In addition, we computed the difference in the runtimes if one of the techniques is faster. We summarize the results in Tab. 4.7 for the dependency analysis and in Tab. 4.8 for the slice computation. In the context of the dependency analysis, our technique is slower in more executions than the standard technique for the initial version θ_0 of the Wiper and Mine Pump SPL as well as for the versions θ_1 and θ_3 of the Vending Machine SPL. This is consistent with the results of the hypothesis test. However, for those cases the runtime of our technique is slightly slower by means of more or less 1 ms. For the remaining versions of all subject systems our technique is faster in 70% to 90% of the executions with an increasing difference in the runtime between both techniques. In the context of the slice computation, our technique is, for all subject systems and their versions, in circa 50% of all executions faster than the standard technique. In contrast to the dependency analysis, for the slice computation, the number of equal runtimes is on average 20% of all executions and also the difference in the runtimes between both techniques is rather small. We conjecture that our technique has a slight advantage in terms of the runtime, but the standard slice computation and slice difference derivation has a similar performance.

To summarize, the obtained results show that our incremental model slicing technique is efficiently applicable for change impact analysis (**RQ2**). Compared to the standard slicing technique, our incremental slicing technique has a better performance for the dependency analysis based on the exploitation of the commonality between subsequent variants to be analyzed (**RQ2a**). For the slicing subprocess, we achieve also better runtimes compared to the standard technique (**RQ2b**), but the difference between the respective runtimes are rather small. However, as the dependency analysis as well as the slice computation are always applied together, we conjecture that our incremental model slicing outperforms the standard slicing technique, especially, if we apply our technique for change impact analysis of larger variant-specific state machines.

RQ3. In Tab. 4.9, we summarize the categorization results of the reasoning process about the higher-order delta application obtained for the three evolving delta-oriented subject SPLs. In the second column, we provide the total number of variants of an SPL version. The remaining columns present the categorization identified based on the application of the incremental delta set derivation. Similar to **RQ1**, we present the data of solely one execution out of the 100 repetitions as our reasoning process provides as expected the same categorization results for the three SPLs and their versions in each repetition. The third column comprises the number of new variants. The fourth column contains the number of modified variants. In the fifth column, we provide the number of unchanged variants. The last column captures the number of removed variants.

Table 4.9: Results of the Reasoning about Higher-Order Delta Application for the **W**iper, **V**ending **M**achine, and **M**ine **P**ump SPL (# = Number)

SPL	#Variants	#New Variants	#Modified Variants	#Unchanged Variants	#Removed Variants
W_{θ_0}	8	8	0	0	0
W_{θ_1}	8	0	8	0	0
W_{θ_2}	12	4	0	8	0
W_{θ_3}	24	12	0	12	0
W_{θ_4}	24	0	12	12	0
VM_{θ_0}	28	28	0	0	0
VM_{θ_1}	42	14	0	28	0
VM_{θ_2}	70	28	14	28	0
VM_{θ_3}	42	0	0	42	28
VM_{θ_4}	42	0	24	18	0
VM_{θ_5}	42	0	24	18	0
VM_{θ_6}	48	18	24	6	12
MP_{θ_0}	16	16	0	0	0
MP_{θ_1}	16	0	8	8	0
MP_{θ_2}	32	16	8	8	0

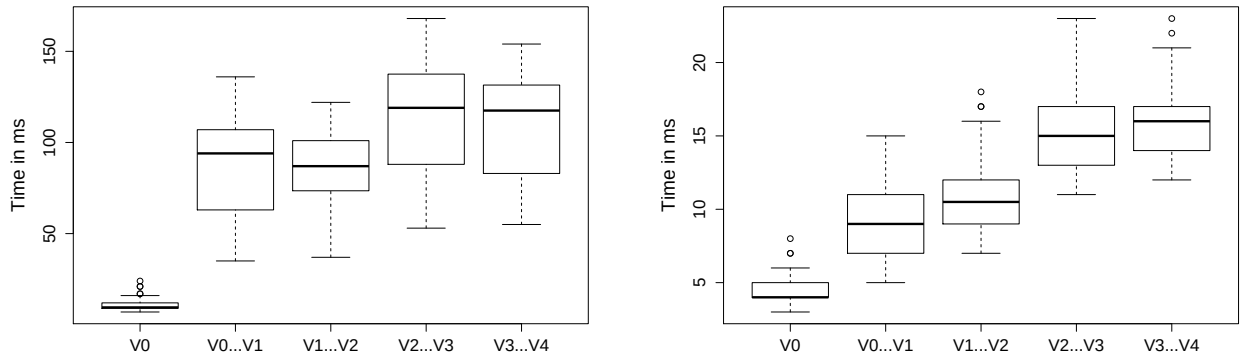
For all SPLs and their versions, the results show that our delta set derivation identifies the same total number of variants as derivable by incorporating the set of feature configurations (cf. Tab. 3.4). Obviously, for the initial versions of the three subject systems, we categorize each variant as new. In contrast, we obtain varying categorizations for the remaining SPL versions which are validated based on the comparison to the original categorization manually defined for the complete documentation of the delta-oriented SPLs and their evolution history [NLS18].

As we can see, the reasoning process results in different scenarios of categorizations such that we identify (1) only modified or unchanged variants, (2) combinations of new and unchanged as well as modified and unchanged variants, or (3) the combination of new, modified, and unchanged variants. In the most version-specific categorizations, we detect unchanged variants. This is an important fact to be exploited by our model-based regression testing framework in order to reduce the overall testing effort when testing subsequent SPL versions (cf. Chapt. 5). For instance, version θ_3 of the Vending Machine SPL represents a special case, where we detect solely unchanged variants. During the evolution step from version θ_2 to θ_3 the functionality of variable beverage sizes is changed by removing one offered size [NLS18]. This removal implies the removal of the deltas which are related to this functionality and, therefore, also the removal of respective variants. As the

other variants of version θ_2 are not influenced by the delta removal and nothing else is changed by the corresponding higher-order delta of the evolution step, those variants remain in version θ_3 as unchanged variants. The removal of variants (12) is also detected in the evolution step from version θ_5 to version θ_6 of the Vending Machine SPL which can be seen by comparing the total number of variants of θ_5 and the number of modified as well as unchanged variants of θ_6. Again, deltas are removed via the higher-order delta such that 12 variant-specific delta sets are not derivable anymore indicating the removal of the respective variants.

In summary, the results of the categorization show the applicability of our variant set change impact analysis (**RQ3**). Based on the reasoning about higher-order delta applications, we identify changes to the variant set in terms of unchanged, modified, removed, and new versions of variants to guide the retest test selection of our model-based SPL regression testing framework (cf. Chapt. 5). Furthermore, the three subject systems and their versions cover distinct categorization scenarios as described above which were detected by our change impact analysis. The impact of the categorizations on our testing framework is evaluated in Sect. 5.4.

RQ4. We investigate the efficiency of our reasoning about higher-order delta application (**RQ4**) by assessing the execution time when applied for the three SPLs and their versions. As described for the research methodology, the runtime of the change impact analysis is a crucial cost factor for regression testing techniques to be efficient according to Leung and White [LW91], which is why we focus on the runtime evaluation and abstract from a distinct evaluation of the size of variant trees by means of the number of tree nodes. As the reasoning process is divided into the subprocesses of incremental delta dependency analysis and incremental delta set derivation, we provide the runtime results also divided for both subprocesses. The results presented and discussed in the following are obtained based on 100 repetitions of a complete execution for each subject system as defined in the research methodology description. For a complete execution, we step from each SPL version $\theta_i \in \Theta$ to its subsequent SPL version $\theta_{i+1} \in \Theta$ of a subject SPL and apply the variant set change impact analysis. In addition, we omit a comparison of the analysis runtimes of our reasoning approach to those of the naive product-by-product approach as the automatic categorization is not fully automatable for the variant-set change impact analysis as already discussed for the research methodology.

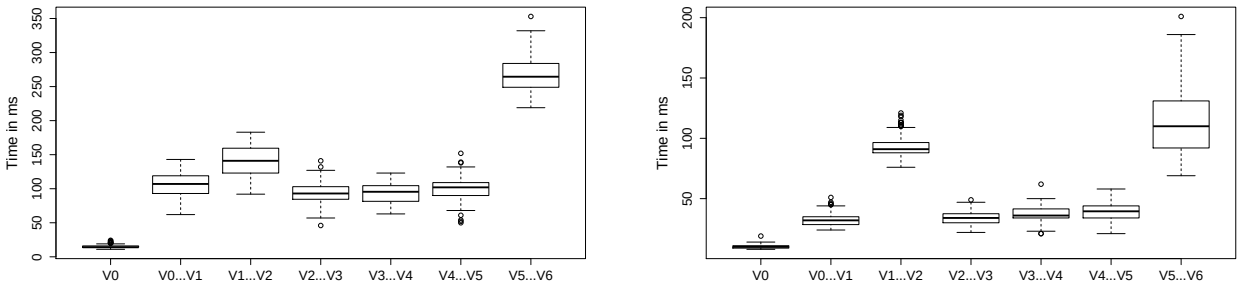


(a) Delta Dependency Analysis

(b) Variant Tree Generation

Figure 4.23: Runtime Results of the Delta Dependency Analysis and Variant Tree Generation Including Delta Set Derivation of the Wiper SPL

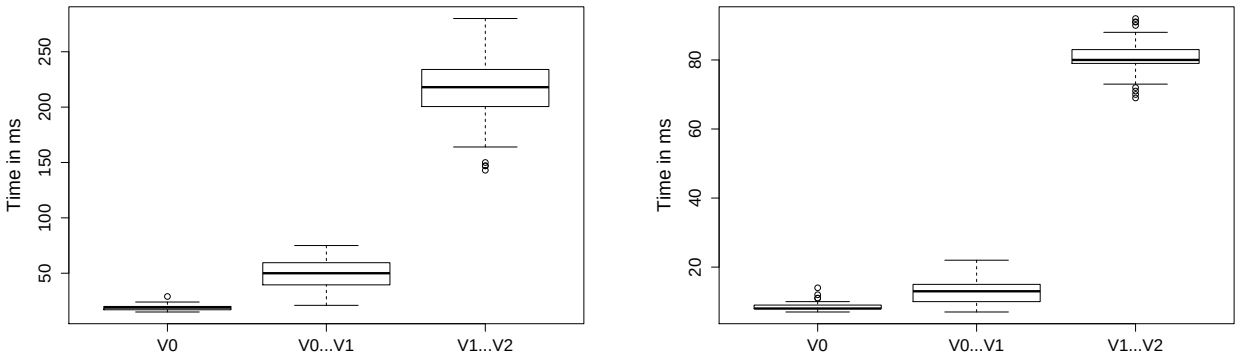
In Fig. 4.23, the runtime results of the incremental delta dependency analysis and variant tree generation for the Wiper SPL are shown in respective box plots. The first box plot represents the distribution of execution times for the initial SPL version θ_0, whereas the remaining box plots denote the analysis time distribution when stepping to the subsequent SPL versions. As we can see, both subprocesses require individually and also combined less than one second for their tasks. In contrast to the incremental dependency graph adaptation of the incremental slicing technique (cf. Sect. 4.1.2), we cannot achieve a reduction of the analysis time by exploiting the commonality as we always have to (re-)check the set of delta dependencies for each delta node. Hence, we require more analysis time for subsequent SPL versions due to the increasing number of deltas (cf. Tab. 3.4) used to derive variant-specific state machine test models. We can see the same relation between number of deltas and execution times for the incremental variant tree generation. Furthermore, the results show that for small evolution steps (i.e., from version θ_1 to version θ_4), where the number of deltas increases by approximately two in each step as documented in Tab. 3.4, the impact on the execution time is likewise small such that similar, yet different execution times are achieved.



(a) Delta Dependency Analysis (b) Variant Tree Generation

Figure 4.24: Runtime Results of the Delta Dependency Analysis and Variant Tree Generation Including Delta Set Derivation of the Vending Machine SPL

In Fig. 4.24 and Fig. 4.25, the runtime results of the incremental delta dependency analysis and variant tree generation for the Vending Machine as well as Mine Pump SPL are shown, respectively. For both subject systems and their versions, we can see the same tendency of runtimes w.r.t. the



(a) Delta Dependency Analysis (b) Variant Tree Generation

Figure 4.25: Runtime Results of the Delta Dependency Analysis and Variant Tree Generation Including Delta Set Derivation of the Mine Pump SPL

number of deltas as for the Wiper SPL. The initial SPL version requires the lowest execution time of the subprocesses. Again, for the remaining versions and, hence, increasing number of deltas comprised in the respective delta models (cf. Tab. 3.4), the execution times increase as well. For both evolving SPLs, we achieve execution times less than one second for the complete reasoning process, i.e., the combined execution times of the two subprocesses.

To summarize, the runtime results show that the reasoning about the higher-order delta application is efficiently applicable to determine the changes of the variant set when stepping to the next SPL version to be tested (**RQ4**). Our reasoning technique requires milliseconds for the three evolving SPLs. Of course, with an increasing number of deltas, those execution times will also increase. However, compared to the time required by other activities performed during SPL testing, the times of our analysis technique will still have a neglecting footprint on the overall testing time.

4.3.3 Threats to Validity

For the evaluation and definition of our change impact analysis techniques, the following threats to validity arise. The selection of the three subject systems may be a potential threat. The selection of an evaluation subject is a general drawback when applying a controlled experiment as it influence the potential to generalize the results for other systems. We selected the three SPLs as they provide distinct evolution and modeling characteristics influencing the evaluation differently as seen in our evaluation results. In addition, the size of the three subject systems may be another potential threat. However, the initial versions of the three SPLs were already applied as benchmark for SPL quality assurance in the literature [Cla10; LMT+16; LNT+19]. Furthermore, their evolution histories were defined following a structured evolution process such that distinct evolution scenarios by means of varying categorizations of variants are covered [NLS18]. Based on the obtained results, we assume that they are, up to a certain extent, generalizable also to other evolving model-based SPLs which uses delta-oriented state machines as (test) modeling formalism. This assumption has to be substantiated by performing more experiments with other evolving delta-oriented SPLs.

The step of delta-oriented test modeling is another potential threat. Due to varying interpretations of requirements and, thus, of a systems' behavior, this step may result in different test models. As the application of our incremental model slicing technique and the reasoning about higher-order delta application is based on a given delta-oriented test model, their application is potentially influenced by the test-modeling step. However, this problem applies in general for model-based testing [UL06; UPL12] and is not a specific threat for delta-oriented test modeling. To cope with this threat, we compared our re-engineered models of the original versions of the subject SPLs with the original documented models [Cla10] to validate that both instances specify the same behaviors and defined the evolution scenarios in a structured way based on the validated models [NLS18].

The limitation to control dependencies of our slicing technique is a potential threat as it restricts the applicability of the impact analysis to event-based systems. Our slicing technique is applied to facilitate change impact analysis in delta-oriented state machine test models which define the abstract behavior of an SPL under test. Based on the abstraction and according to Milner [Mil89], we are able to encode the read-/write-access of variables via events such that also complex behavior is specifiable based on events. This encoding allows for the weakening of the potential threat of the restricted applicability. For the general application of our slicing technique to other delta-oriented SPLs, an extension of the dependency analysis to also incorporate data dependencies is conceivable.

As already mentioned, the order in which variants are subsequently analyzed by applying our incremental slicing technique for change impact analysis may influence the detection of slice differences and, hence, retest potentials. To cope with this potential threat, we subsequently analyzed each variant with the remaining variants of an SPL to be independent from an order of variants during the evaluation. We will perform a controlled experiment regarding the influence of testing orders on our change impact analysis in Sect. 5.4.

To exclude external influences from the runtimes of our change impact analysis techniques, e.g., based on the computational bias of the operating system, we performed each application of our techniques 100 times for each of the three subject SPLs and their versions. Based on the 100 repetitions, we further facilitate the statistical interpretation of the obtained results.

We abstracted from a distinct evaluation of the size of variant trees in terms of number of tree nodes as we focused on the runtime of the variant set change impact analysis to reason about the efficiency of the impact analysis. Nevertheless, the size of variant trees and their exponential growth w.r.t. the number of deltas may be a potential threat. As we evaluated the runtime which also depends on the size of the variant tree to be analyzed, we implicitly evaluated the tree sizes. However, the real influence of the size of variant trees should be evaluated by performing more experiments with other evolving delta-oriented SPLs. In this context, an improvement of the prototype and variant tree data structure is conceivable to improve the memory footprint as well as the data access of the information captured in a variant tree.

For the reproducibility of our evaluation and the obtained results, we provide our prototypical implementation, the delta-oriented test models, and all data gathered during the controlled experiment online `https://github.com/SLity/mbtSPLregression`.

4.4 Related Work

We discuss related work w.r.t. (1) variability-aware slicing, (2) incremental slicing, (3) change impact analysis applied in the context of SPLs, and (4) slicing-based change impact analysis. For a general overview of state-based model slicing techniques, we refer to the survey of Androutsopoulos et al. [ACH+13]. In addition, we refer to Bohner [Boh96] and Arnold [Arn96] for a general discussion about the importance of impact analysis as well as to Lehnert [Leh11b; Leh11a] for a taxonomy and survey of change impact analysis techniques for single-software systems.

4.4.1 Variability-Aware Slicing

In the context of SPLs, slicing is proposed in the problem space for feature models [ACL+11c; KST+16; AKT+16b] and in the solution space for annotative source code [KS14; AGP+15] as well as state machines [KLB12] and for deductive verification of delta-oriented SPLs [BKS11].

Problem-Space Slicing Techniques. **Acher et al.** [ACL+11c; ACL+11a] proposed feature-model slicing to cope with the increasing complexity of large-scale feature models. Based on a set of features used as slicing criterion, their technique decomposes and, hence, reduces a given feature model such that the resulting slice solely contains the input features and all features that are related to them via constraints. They integrated their technique in the FAMILAR environment [ACL+11b] to facilitate an efficient management of large-scale feature models. **Krieter et al.** [KST+16] introduced another technique for feature-model slicing to be used, e.g., to identify feature model interfaces [SKT+16] for compositional feature-model analysis. They realize the slicing of a feature model represented as

conjunctive normal form based on logical resolution and minimization of logical formulas. **Ananieva et al.** [AKT+16b] adopted the slicing technique of **Krieter et al.** [KST+16]. They apply their technique for the identification of implicit cross-tree constraints and further proposed a technique for the derivation of explanations for identified implicit constraints based on boolean constraint propagation. Compared to our incremental model slicing technique, those techniques are applied to feature models supporting analysis tasks in the problem space and do not allow for solution-space change impact analysis as required by our model-based SPL regression testing framework.

Solution-Space Slicing Techniques. **Kamischke et al.** [KLB12] proposed conditioned model slicing for annotative state machines. They extended the work of **Ji et al.** [JWZ02] by incorporating the variability information during the slice computation. Based on a slicing criterion defined as a tuple of a state machine element and a (partial) feature configuration, their technique reduces the annotative state machine by abstracting from those elements which do not influence the element and do not satisfy the (partial) feature configuration of the given slicing criterion. In contrast to our state machine slicing technique, where we use delta-oriented state machines as input, their technique is applied to annotative state machines. In addition, their technique facilitates the analysis of state machines to support, e.g., model comprehension, but does not allow for change impact analysis which is required to support SPL regression testing. However, we exploit the control dependencies defined by **Kamischke et al.** [KLB12] in order to realize our incremental slicing technique.

Kanning and Schulze [KS14] introduced variability-aware program slicing for annotated C programs. They extended program dependence graphs to also incorporate variability information introduced by C preprocessor directives. During the slice computation, their technique abstracts from statements that do not influence a given slicing criterion similar to standard program slicing [Wei81; Tip95]. Furthermore, they incorporate the determined variability information into the slice to detect and reason about potential feature interactions. **Kanning and Schulze** [KS14] integrated their approach in the TypeChef [LvRK+13] research infrastructure, i.e., a framework for the analysis of annotated source code. Compared to our approach, their technique is solely applicable to annotated C programs. In addition, the technique is not capable to explicitly specify differences between slices of subsequent variants to reason about the change impact.

Angerer et al. [Ang14; AGP+15] presented a configuration-aware analysis technique for estimating the effort of implementing a change request based on the application of program slicing of annotated source code. They extended system dependence graphs to define a conditional system dependence graph by incorporating presence conditions representing compile- and load-time variability information. Based on the conditional system dependence graph, their technique first apply backward slicing starting in the graph nodes mapped to statements that should be modified to determine the presence conditions specifying under which configuration options the statement can be executed. Afterwards, they apply forward slicing to identify the parts of the software and the variants that are potentially affected by the modification to be made exploiting the determined presence conditions. **Angerer et al.** [APG16] improved their technique to become modular, i.e., they are able to compose pre-computed analysis results to determine the impact of a potential modification. In contrast to **Angerer et al.** [Ang14; AGP+15], our slicing technique is model-based and focuses on a different application scenario of change impact analysis. Our technique identifies changed dependencies between subsequently tested variants and version of variants captured as slice differences to reason about the change impact and, therefore, to guide retest test selection, whereas their tech-

nique is applied to support the maintenance task of implementing change requests by providing a developer with the information which variants and software parts will be influenced by the change.

Bruns et al. [BKS11] proposed delta-oriented slicing applied in the context of deductive SPL verification. Based on a given set of proofs for the core variant, delta modules that transform the core into another variant are analyzed to identify whether the encapsulated changes have an influence on the existing set of proofs. The result of this analysis is a delta-oriented slice capturing all proofs that have to be redone for the new variant under consideration. Compared to our slicing technique, **Bruns et al.** [BKS11] use a different notion of a slice and also focus on a different application scenario as regression testing for their analysis, i.e., impact analysis to support deductive verification.

4.4.2 Incremental Slicing

Existing techniques for incremental slicing were proposed to support debugging [OSH01], verification [Weh06], and model-based development [POK+17].

Orso et al. [OSH01] presented an approach for incremental program slicing to improve software debugging and program comprehension. Their approach requires the program to be sliced and a selection of a set of data dependencies to compute a static slice. The slice is used to observe a program property. In case the slice is insufficient for the observation, the set of data dependencies is extended resulting in the incremental adaptation of the initial slice based on the incorporation of the new dependencies. This step is repeated until the observation of the property is achieved. Compared to **Orso et al.** [OSH01], our slicing technique focuses on control dependencies for the slice computation. In contrast to the incremental adaptation of the same slice, we incrementally recompute a slice for a slicing criterion contained in subsequently tested variants to detect changed execution dependencies which are captured as slice differences indicating retest potentials.

Wehrheim [Weh06] proposed an incremental slicing technique to reduce the effort for software verification. For a property to be verified given as temporal logic formula, an initial simple slice is computed by applying abstraction refinement representing an overapproximation of the behavioral specification of the system. If the property holds for the initial slice, the property also holds for the complete system. In contrast, if a counter example is found in the initial slice, the example is validated against the complete specification. In case the counter example is not valid for the complete specification, i.e., the example does not proof the failing of the property, the slice is incrementally refined by relaxing the data abstraction until the given property can be verified or a real counter example is found. Compared to **Wehrheim** [Weh06], our slicing approach focuses on the identification of retest potentials by means of changed execution dependencies between slices for the same slicing criterion of subsequently tested variants or version of variants and do not incrementally adapt the same slice. In addition, our incremental slicing technique is applied for change impact analysis to support model-based regression testing.

Pietsch et al. [POK+17] presented an incremental slicing technique to support large-scale model-based development. Their technique computes a slice of a given large model based on the selection of model elements to be contained. To ensure the consistency of the slice which is defined by the underlying meta model of which the orginal model and, therefore, the slice are instances of, further elements are added to the slice. The slice represents a submodel which can also be interpreted as a certain view on the large model. In case, the initial selection of model elements is changed by adding or removing elements, the slice is incrementally adapted by adding or removing those elements

and also the elements which are related to the selected elements to reensure the consistency of the slice. They exploit the graph transformation framework Henshin[7] to apply model differencing and patching for the slice computation and incremental adaptation. **Taentzer et al.** [TKP+18] extended the work of **Pietsch et al.** [POK+17] by proposing the formal foundation of incremental model slicing for graph-based models using model modifications. Compared to **Pietsch et al.** [POK+17], our slicing technique is applied as change impact analysis by identifying slice differences indicating retest potentials between subsequently tested variants and version of variants, whereas their technique is used to provide a certain view of a large model for a developer. Furthermore, our technique is based on control dependencies that are defined via the execution semantics of our state machine test modeling formalism, whereas their technique is based on the syntax of the model by incorporating the underlying meta model to ensure model consistency.

4.4.3 SPL Change Impact Analysis

In the context of SPLs, different techniques for change impact analysis were proposed for reasoning about the result of SPL evolution [DKvD+14; MBB16; PDŠ12; SKT+16; HRG12; HGB+18; DPG+11; YM12; SK14; TBK09; BKL+16; NSS16; QPB+14; PYZ11; DNG+08b; NBA+15; SBT16]. In contrast to those techniques which are discussed in the following paragraphs, our reasoning about higher-order delta application facilitates change impact analysis in the solution space by determining the changes to the variant set in terms of added, removed, modified, and unchanged variants. Furthermore, none of those techniques is applied to support SPL regression testing and, therefore, focus on different application scenarios of change impact analysis in the context of evolving SPLs.

Change Impact Analysis for Feature-Model Evolution. **Dintzner et al.** [DKvD+14] introduced feature-based change impact analysis to determine how existing configurations of a multi product line are affected by feature-model evolution. Their technique identifies the set of variants which are no longer derivable due to a change of the feature model by propagating the feature change to the set of existing variants. Based on this analysis, they provide a domain engineer the information which configurations of the multi product line are affected by the change and should either be removed or be adapted to correspond to the new feature model and, hence, SPL version. They extended their work by realizing the tool FEVER to also incorporate the modification of build systems and source code for their change impact analysis [Din17].

Maâzoun et al. [MBB16] proposed change impact analysis for feature model and design asset evolution to support the estimation of the effort required for integrating a requested change. Their technique uses the traceability between source code and features of a feature model to identify the variants as well as parts of the implementation that would be affected by the change. For establishing the traceability, they re-engineer the feature model from the source code. The analysis result supports a developer or maintainer to decide whether the change request can be integrated or is dismissed as the effort would be too high.

Paskevicius et al. [PDŠ12] presented a change impact model to facilitate the reasoning about the changeability of a feature model and, hence, the validity of a change to be made during feature model evolution. The change impact model is based on a feature dependency matrix, where changes of features can be propagated to related features. By comparing the original feature model and its new version, where the information determined by the change impact model is taken into account,

[7] https://www.eclipse.org/henshin/, last access: May 31st, 2019

they determine values such as the number of change-affected features or number of change-affected variants using the Jaccard distance to examine the changeability of a feature model w.r.t. the change to be made. In case the changeability values are low, the change to be made should be rejected.

Schroeter et al. [SKT+16] realized a change impact analysis for feature model interfaces when feature model evolution occurs. Feature model interfaces facilitate the compositional analysis of feature models. Each interface represents a decomposition, i.e., a subpart of a feature model which can be analyzed individually. By composing the individual analysis results, the analysis of the complete feature model is achieved more efficiently. In case the feature model is changed due to an evolution step, their technique identifies the feature model interfaces which are affected by the change and, therefore, have to be re-analyzed.

Heider et al. [HRG+12] proposed a change impact analysis for evolving SPLs by exploiting the concepts of regression testing which is integrated in the Dopler framework [DGR+10]. After a modification of the feature model of an SPL under consideration, they identify the set of variants which are used as test cases that have a difference in their feature configuration w.r.t. the previous SPL version. The change information of variants is then propagated to the domain engineer to reduce the error potential when evolving the variability model to correspond to, e.g., changing requirements.

Sabouri and Khosravi [SK14] proposed two change impact analyses to support regression verification of evolving SPLs. The first analysis identifies unchanged variants after the evolution of a feature model, where they examine the applied changes to reason about the equality of feature configurations without generating each configuration anew. The second analysis takes the changes of the source code into account to identify the set of properties which have to be re-verified. They apply static program slicing to identify those statements which are influenced by the modified statement captured in a respective slice. Based on the slice, the set of already verified properties is examined to determine those properties which depends on the execution of a statement contained in the slice. For those properties, a re-verification has to be performed.

We already discussed techniques for feature model evolution in Sect. 3.5 which also defines SPL change impact analysis. In the following paragraphs, we shortly recap those analysis techniques.

Thüm et al. [TBK09] reason about feature model edits and their impact on the variant set. They compare the original and the evolved feature model and detect the applied change operations exploited to compute a change classification w.r.t. the impact on the set of feature configurations in terms of refactorings, specializations, generalizations, or arbitrary edits by using a constraint solver.

Bürdek et al. [BKL+16] proposed as similar approach based on the reasoning about feature model edits. They detect changes between two versions of a feature model and document them as a sequence of (complex) edit steps. This sequence is then exploited to compute the semantical difference between both feature model versions w.r.t. the set of derivable variants, where the classification of **Thüm et al.** [TBK09] comes into effect.

Nieke et al. [NSS16] introduced a change impact analysis technique for temporal feature models. When stepping to the next SPL version, they reason about the impact on feature configurations based on a catalog of (atomic/complex) feature model evolution operations and their defined application semantics. The analysis result supports SPL developers in order to guarantee that the evolution does not affect certain feature configurations of interest, i.e., the configurations remain valid during the evolution step. In addition, **Nieke et al.** [NST18; NMS+18] defined an evolution

anomaly detection and explanation technique based on temporal feature models. The technique identifies inconsistencies, the operations that lead to them, and the respective evolution step by investigating the complete evolution history, e.g., to incorporate intermediate evolution steps that do not violate the validity of already planned evolution steps.

Quinton et al. [QPB+14] discussed common change operations applied to evolve cardinality-based feature models and their relation to resulting inconsistencies. Both, the set of change operations and the information about their impact, are exploited to realize an automated detection and explanation of inconsistencies during the evolution of cardinality-based feature models.

Change Impact Analysis for SPL Requirements Evolution. **Hajri et al.** [HGB+18] presented an analysis approach to examine the impact of the evolution of configuration decisions to support use-case-driven development of evolving SPLs. Their approach identifies which prior and subsequent configuration decisions and, therefore, variants are affected by a decision modification. The result is afterwards used to incrementally regenerate variant-specific use case models which are related to affected configuration decisions.

Díaz et al. [DPG+11] introduced impact analysis to estimate the effort required to implement a change in the requirements of an SPL. Based on a change to be made in the product-line requirements, their technique first identifies related design decisions which are affected by the potential change and afterwards follows existing traceability links between requirements, design decisions, and elements of the product line architecture to also determine the set of change-affected architectural elements. For the detection of the complete set of affected architectural elements, they also take the dependencies between elements of the product-line architecture into account and add those elements to the impact set that are related to already added elements. The resulting impact set is provided to the developer to reason about the required effort for the change implementation.

Peng et al. [PYZ11] presented a change impact analysis to support planning and risk management of SPL evolution. As discussed in Sect. 3.5, they focused on the evolution of the SPL requirement specification. To analyze the impact on the variability defined by the features of the feature model, they exploit traceability links between requirements and features.

Change Impact Analysis for Solution-Space Evolution. **Yazdanshenas and Moonen** [YM12] presented a technique for change impact analysis to support the maintenance of evolving component-based SPLs. They define a family-wide dependence graph to capture dependencies between the source code of components and configuration artifacts, e.g., features. Based on planned changes to be made in the source code of a component the respective component interface is the starting point of the impact analysis. Their technique uses static program slicing incorporating the family-wide dependence graph to identify the final impact set of modification-affected components. The resulting impact set is ranked to provide a scale of the change impact and given to the maintainer in order to plan and perform the change implementation.

Dhungana et al. [DNG+08a; DNG+08b; DGR+10] defined a change impact analysis technique applied in the Dopler framework for inconsistency detection after an evolution step. As discussed in Sect. 3.5, in the Dopler framework, model fragments are used to specify domain artifacts and their interdependencies. The impact analysis check the modified as well as interrelated model fragments whether the made change has introduced inconsistencies.

Neves et al. [NBA+15] proposed an impact analysis approach by reasoning about the application of safe evolution templates which were discussed in Sect. 3.5. Their analysis checks whether an

applied template, i.e., the addition of behavior, results in a refinement and, hence, in a preservation of already verified behavior. In addition, **Sampaio et al.** [SBT16] extended the work by facilitating the analysis of partial safe evolution templates, where they also incorporate modifications and removals as evolution scenarios. They verify for behavior preservation for those variants which do not contain a modified or removed artifact. Therefore, the analysis results are usable to reason about change impact to provide an SPL developer the information which variants are not affected by changes.

4.4.4 Slicing-Based Change Impact Analysis

Existing slicing techniques used to facilitate change impact analysis in the context of single-software systems are applied to support (1) white-box and model-based regression testing by identifying the influence of a change to already tested parts of the software under test [AHK+93; JG06; BH93; RH94; GHS96; Bin97; Bin98; TLS+10; PM10; LM08; OAH03; GL91; KTV02; CPU07a; UY13], or (2) maintenance tasks, i.e., the integration of a change request, by allowing for an estimation of the required effort [AR11; HRH+05; Zha02; ZYX+02; FM06]. In contrast to those techniques, we apply incremental model slicing for change impact analysis to support model-based regression testing of evolving SPLs. We refer to Binkley [Bin98] for an overview of program slicing applied for change impact analysis and to Li et al. [LSL+13] for an overview on code-based change impact analysis in general.

Slicing for Supporting Regression Testing. **Agrawal et al.** [AHK+93] introduced three types of dynamic slices that comprise the executed statements for a test case. For instance, an execution slice captures all statements that are traversed during a test case execution, whereas a relevant slice solely contains those statements that directly and also indirectly influence the output of the test case. A retest of a test case is required if at least one statement in a slice is modified. In addition, they provide a discussion regarding the application scenarios for their types of slices as each type allows for a different abstraction of statements w.r.t. the execution of a test case and, therefore, for a different impact analysis of a source code modification. **Jeffrey and Gupta** [JG06; JG08] exploited the technique of **Agrawal et al.** [AHK+93] and further incorporated test-case prioritization to improve the retesting of test cases. The weight of a test case which is taken into account for the prioritization depends on the number of modified statements as well as the total number of statements comprised in the respective relevant slice. On the contrary, our slicing technique focuses on the static dependencies and their changes, i.e., indicated by slice differences, for a given point of interest, e.g., a transition. Based on the static analysis, we are able to determine the complete impact of changes applied to the state machine test model and are not restricted to test-case-specific slices, where the quality of the test suite by means of coverage is a crucial factor. In contrast to **Jeffrey and Gupta** [JG06; JG08], we perform solely retest test selection. However, test-case selection and prioritization are independently applicable and can also be combined for efficient and effective regression testing [YH12].

Bates and Horwitz [BH93] proposed static slicing on program dependence graphs and examined slice isomorphism to reason about the retest of test cases. In addition, they define test adequacy criteria for program dependence graphs such as all-control-nodes or all-control-edges used to guide the slicing process. For instance, for each control edge, their technique computes a slice and this slice is recomputed after a modification. If a control node, e.g., a statement or function, contained in both slices has a changed set of incoming or outgoing edges, the node was modified or is influenced by a modification. In such cases, their technique selects respective test cases to be retested. The comparison of slices to identify changed dependencies is similar to our approach, but, in con-

trast to **Bates and Horwitz** [BH93], we determine the slice differences during the incremental slice computation.

Binkley [Bin97] introduced two slicing techniques to allow for change impact analysis in the context of white-box regression testing. The first technique determines the semantical differences between the original and modified program during their execution, i.e., the slice contains the behavior of the modified program that differs to the original program. The second slicing technique extends the work of **Bates and Horwitz** [BH93] such that interprocedural programs can be sliced based on the extension of program dependence graphs called system dependence graph. Both techniques combined allow for the detection of the change impact to guide retest test selection and to facilitate the execution of the retest test suite on the reduced program solely comprising the semantical differences. As **Binkley** [Bin97] exploits the impact analysis of **Bates and Horwitz** [BH93], our slicing technique is, again, similar regarding the detection of retest potentials. The execution of the retest test suite on the reduced program is a benefit which our technique cannot provide as we apply our change impact analysis to variant-specific test models and, hence, on the specification level.

Rothermel and Harrold [RH94] proposed a change impact analysis technique that is similar to forward slicing. To detect retest potentials to be retested by selecting test cases for reexecution, their technique traverses the program dependence graph of the original and modified program. In case a difference is detected indicating the influence of a source code modification, they select all test cases that traverses the modified graph node during its execution. In addition, their technique identifies def-use pairs which are affected by modifications and also select test cases to retest the respective behavior. If an affected def-use pair is not yet covered by an existing test case, they generate a new one such that the respective behavior can be retested. Compared to **Rothermel and Harrold** [RH94], our technique incorporates solely control dependencies due to the focus of event-based test models and apply backward slicing to capture all state machine elements that influence a given criterion during execution to reason about changed dependencies when the original state machine test model has changed. However, our model-based regression testing framework also generates new test cases if the retest potentials derived based on slice differences are not covered by the current test suite of a variant or version of a variant as described in Chapt. 5.

Gupta et al. [GHS92; GHS96; HS88] also applied program slicing as change impact analysis to detect affected def-use pairs. Depending on the source code modification, e.g., the usage of a variable in a statement is modified, they apply backward slicing to determine statements that define a new valuation for the variable. In case the definition of a variable is changed, they apply forward slicing to obtain the respective statements where the variable is used. In addition to the direct identifiable def-use pairs which are affected by a change, their technique also identifies indirectly affected def-use pairs, i.e., the definition or the computational use of a variable is dependent from the variable of a directly affected def-use pair. For all affected def-use pairs, they select test cases to be retested. Again, our technique incorporates solely control dependencies and apply backward slicing.

Gallagher and Lyle [GL91] proposed a slicing technique for decomposing a program to support change impact analysis and regression testing. Based on a given variable used as slicing criterion, their technique creates a decomposition as well as a complement slice. The decomposition slice comprises all statements that are related to the slicing criterion, whereas the complement slice contains the non-related statements of the program. This division supports a developer or maintainer such that the modification has solely be performed in the decomposition slice. Furthermore,

the impact analysis as well as the retesting are solely applied on the decomposition slice. To obtain the complete program after modification and retesting, the decomposition slice is merged with the complement slice which was not affected by the modification. In contrast to **Gallagher and Lyle** [GL91], our slicing technique does not provide a decomposition, but directly facilitates change impact analysis to guide the subsequent process of retest test selection.

Tao et al. [TLS+10] introduced the application of slicing to allow for change impact analysis for object-oriented programs. They include the logical hierarchy, e.g., from package to statement level, of object-oriented software, e.g., developed based on Java, in their slicing technique. Starting in a modification, they apply backward and forward slicing to identify the impact of the modification in the distinct hierarchy levels. Based on the analysis results, they select test cases from the different affected hierarchy levels for retesting the influenced behavior that was already tested. Compared to **Tao et al.** [TLS+10], our slicing technique solely applies backward slicing and focuses on the test model level, i.e., state machine level. Furthermore, our technique determines slice differences that indicate retest potentials, whereas their slicing technique determines no differences to the original program such that the complete behavior which is captured by a slice has to be retested.

Panigrahi and Mall [PM10] applied program slicing to facilitate impact analysis for regression testing of object-oriented programs. They capture control and data dependencies in an extended object-oriented system dependence graph, where object-oriented relations are also incorporated, e.g., inheritance and aggregation. Based on this graph, they apply forward slicing to identify all change-affected graph elements, for which they select respective test cases to be reexecuted. In addition, **Panigrahi and Mall** [PM10] determine the coverage of elements which are indirectly retested, i.e., elements traversed during the execution of a reexecuted test case. Based on the coverage of change-affected and indirectly tested elements, the selected test cases are further prioritized. In contrast to **Panigrahi and Mall** [PM10], our slicing technique is defined for state machine test models and solely incorporates control dependencies. Furthermore, we apply backward slicing to identify changing execution dependencies, whereas their technique applies forward slicing to identify elements which are potentially influenced by a change. Both techniques perform retest test selection, but **Panigrahi and Mall** further exploit the results of the impact analysis for prioritization.

Lalchandani and Mall [LM08] adapt the technique of **Binkley** [Bin97] for architecture-based regression testing. Based on a component dependence graph of the original and modified architecture, a slice w.r.t. a given component service is dynamically computed to determine the semantical difference between both architecture versions. The resulting slices are used for reasoning about change-influenced components for which test cases have to be selected for a retest. Compared to our static slicing technique, **Lalchandani and Mall** [LM08] apply dynamic slicing on software architectures. In addition, our SPL regression testing framework defines retest test selection on the component testing level, whereas they define their selection for service-oriented integration testing.

Orso et al. [OAH03] proposed an approach for slicing-based change impact analysis, where a dynamic slice is computed based on the exploitation of field execution data. Field execution data represents real execution traces of the previous program version recorded during the execution by a customer. In case such a dynamic slice comprises program parts that are modified, the respective behavior has to be retested by selecting respective test cases. Their technique furthers allows for the estimation of the effort for customer-specific change implementations by taking their user-profiles, i.e., a subset of the field execution data, into account. In contrast to **Orso et al.** [OAH03], we per-

form static slicing to facilitate change impact analysis and do not gather information for the effort estimation of a change implementation.

Korel et al. [KTV02] proposed dependency analysis to facilitate retest test reduction for model-based regression testing of single-software systems. Their technique apply model differencing to obtain the modifications between two versions of a state machine test model. Those modifications are used to derive test cases to be executed for retesting the modified behavior. To reduce the effort for regression testing by means of less test cases to be executed, they apply dependency analysis w.r.t. the execution path of a test case in a test model. As result of the dependency analysis, for each test case, they provide an interaction pattern which can be interpreted as slice capturing all transitions of a state machine test model that are traversed during the execution of a test case and all transitions which are dependent to those traversed transitions. In case test cases cover the same interaction pattern, they are equivalent and solely one has to be selected for execution. **Korel and Tahat** [KT04] adapted the dependency analysis and applied it as change impact analysis to understand the impact of a change implementation. Again, they first determine the differences between the original and modified state machine model. Afterwards, they determine for each identified modification, the set of transitions that are affected by the modification or that affects the execution of the modification based on their dependency analysis. In contrast to **Korel et al.** [KTV02; KT04], our technique can exploit the explicit specification of differences between variant-specific state machine test models captured as deltas. In addition, we compute slices for transitions and identify changes to their execution dependencies due to modifications, whereas their dependence analysis is applied either for detected modifications [KT04] or for the execution test model path of a test case [KTV02]. The latter scenario also results in the identification of different retest potentials compared to our technique and, therefore, in a different selection of test cases to be reexecuted.

Several adoptions of the work of **Korel et al.** [KTV02; KT04] exist in the literature. **Almasri et al.** [ATK17] extended the impact analysis of **Korel and Tahat** [KT04] to quantify the impact of a modification to support maintenance. Based on a change request, they determine a starting and extending impact set comprising all transitions of a state machine which are dependent on a modification to be performed and compare the sizes of both sets w.r.t. the size of the original model to reason about the potential effort to implement the change request. **Chen et al.** [CPU07a] extended the set of dependencies incorporated in the dependency analysis and also the types of changes to be considered for the analysis. They used the extensions to define a retest test-suite reduction technique [CPU07b] similar to **Korel et al.** [KTV02] and also an approach for retest test-case generation [CPU07a]. Again, compared to our change impact analysis, the dependency analysis results in the identification of different retest potentials and, hence, in a different selection of test cases to be executed for retesting modification-influenced behavior. **Tahat et al.** [TKH+12] presented a technique for model-based retest test prioritization based on the application of the dependency analysis and the execution results of an existing test suite on the modified state machine test model. They further extended their work [TKK+17] in the way that they can apply their prioritization technique also for changes which are solely applied to the source code. In this scenario, they exploit the existing traceability between source code and test model elements and, hence, identify the modified parts in the test model to be retested. In contrast to **Tahat et al.** [TKH+12; TKK+17], we perform retest test selection and also do not have the traceability information between source code and test model. **Ural and Yenigün** [UY13] proposed a similar dependency analysis technique as **Korel**

et al. [KTV02] to facilitate retest test selection which also results in the identification of different retest potentials compared to our technique and, therefore, in a different selection of test cases to be reexecuted.

In contrast to the application of slicing to support white-box and model-based regression testing, **Wehrheim** [Weh05] proposed the use of slicing and dependence graphs of formal specifications as defined by **Brückner and Wehrheim** [BW05] for change impact analysis in the context of regression verification. Forward slicing is applied to identify changes of dependencies of an already verified system property. In case no changes are detected, the verification result is re-used. Otherwise, a re-verification of the property has to be performed. Compared to **Wehrheim** [Weh05], our slicing technique performs backward slicing on state machines and is applied in the context of model-based regression testing. However, both techniques share the same interpretation regarding retest or re-verification potentials in case no changed dependencies are detected.

Slicing for Supporting Maintenance. Slicing-based change impact analysis is also applied in the context of maintenance tasks to allow for an estimation of the effort when a change request should be integrated in the software system. **Acharya and Robinson** [AR11] evaluated static program slicing in an industrial context and based on their investigation they came up with a new approach that can scale for industrial code bases. They apply static forward slicing to support the developer with the information which parts of the code base would be influenced by a change. **Hassine et al.** [HRH+05] proposed change impact analysis on the requirements level based on slicing of use case maps. They also apply forward slicing starting in the modified use case to identify those usage scenarios that will be affected by the change. **Zhao** [Zha02] presented program slicing for aspect-oriented software evolution, where they adapt the technique of **Bates and Horwitz** [BH93] by integrating aspects in the system dependence graph to assess the potential impact of a change. **Zhao et al.** [ZYX+02] further proposed a slicing technique of software architectures to facilitate such kind of change impact analysis to support maintenance also in component-based systems. **Feng and Maletic** [FM06] also allow for change impact analysis to understand changes in software architectures by slicing component interaction scenarios defined as sequence diagrams.

Compared to our slicing technique, all those approaches have a different application scenario and mainly apply forward slicing. In addition, none of those techniques is applicable for state machine test models to identify retest potentials between subsequently tested variants and versions of variants represented as slice differences.

4.5 Chapter Summary

Change impact analysis is crucial for regression testing to guide retest test selection [YH12]. For the definition of our model-based SPL regression testing framework (cf. Chapt. 5), we, therefore, require respective analysis techniques (1) to detect the impact of test model changes to already tested behavior between subsequently tested variants and version of variants by means of changed (inter)dependencies, and (2) to determine the impact of an evolution step to the set of variants in terms of new, removed, unchanged, or modified variants. To this end, we proposed two change impact analyses for an automated reasoning about retest potentials between consecutively tested variants as well as between product-line versions under test.

First, we introduced incremental model slicing for the automated identification of changed execution dependencies, i.e., behavior potentially influenced by changes to the test model between

subsequently tested variants and also between modified variant versions. Therefore, we combined state-based model slicing and delta modeling to exploit the explicit specification of the commonality and differences between variants. By focusing on the differences, we incrementally compute a slice for a given slicing criterion contained in the previous as well as next variant to be tested and determine slice differences, i.e., changed execution dependencies, indicating retest potentials to be retested by our regression testing framework.

Second, we defined the reasoning about the application of higher-order deltas to identify the changes of the variant set when stepping to the next product-line version under test. Therefore, we presented an incremental delta set derivation that is independent from feature configurations based on delta dependency analysis that directly pass on the changes captured in a higher-order delta to be applied to the set of derivable variant-specific delta sets. Our technique automatically examines the changed variant-specific delta sets and their mapping to variant versions to deduce their respective categorization in terms of added, removed, modified, and unchanged variants.

We prototypically implemented our change impact analysis techniques and evaluated their application to the three evolving model-based SPLs (cf. Sect. 3.4). For incremental model slicing, the results show its applicability as change impact analysis as our technique identifies slice differences if they exist. Furthermore, based on the exploitation of the commonality between subsequently tested variants, the detection of slice differences is performed efficiently. For the reasoning process about higher-order delta application, the evaluation results also show the applicability as efficient change impact analysis. The efficiency of both techniques is a benefit as the resulting efficiency of our retest test selection approach for testing variants and versions of variants is influenced by the analysis runtimes according to the cost model of Leung and White [LW91]. We propose the retest test selection which is guided by our change impact analyses and its evaluation in the next chapter.

5 Model-Based Regression Testing of Variants and Versions of Variants

The content of this chapter shares material with work published in [LLS+12], [LMT+16], [LAT+17], and [LNT+19].

Contribution

We propose retest test selection for model-based regression testing of variants and versions of variants. We combine our delta-oriented test modeling and change impact analyses to define a framework for model-based product-line regression testing. The framework facilitates the incremental testing of consecutive product-line versions, where test artifacts and test results of preceding testing processes are reused. To exploit the reuse potential between variants and versions of variants to be tested, we incorporate the results of our impact analyses to automatically select test cases to be retested. Based on this selection, we tackle the potential redundancy during testing of evolving software product lines facilitating the reduction of the overall testing effort. Furthermore, we introduce a technique for computing reuse-optimized testing orders such that our slicing-based impact analysis and the retest test selection benefit for the incremental testing of consecutive variants under test. We prototypically implement our model-based SPL regression testing framework and evaluate its efficiency and effectiveness based on the three evolving model-based software product lines.

In this chapter, we introduce the model-based regression testing framework for testing evolving SPLs. The combination of model-based [UL06] and regression testing [YH12] facilitates (1) the automatic generation of test cases based on state machine test models representing the behavioral specification of a variant under test, and (2) the reduction of test cases to be (re-)executed for testing a variant controlled by applying change impact analysis. Our framework unites delta-oriented test modeling and change impact analyses as well as retest test selection for incremental testing of evolving SPLs. The incremental testing workflow exploits the reuse potential of test artifacts and test results for testing variants and versions of variants subsequently based on preceding testing processes. We tackle the potential testing redundancy emerging due to the shared commonality between subsequent variant versions under test to reduce the overall effort for testing evolving SPLs.

For the guidance of the retest test selection, we propose a retest coverage criterion based on the results of the application of incremental model slicing as change impact analysis between consecutively tested variants and versions of variants. We derive retest test goals by taking the detected slice differences into account representing changed execution dependencies to be retested. To ensure the coverage of the derived retest test goals, we select reusable test case for their reexecution to validate that already tested behavior is not influenced other than intended when stepping to the next variant or variant version under test. If we cannot ensure retest test coverage based on the set of reusable test case, we explicitly generate new test cases for the set of uncovered retest test goals.

Furthermore, the application of incremental model slicing as well as the followed retest test selection depend on the differences between consecutively tested variants captured by means of regression deltas. Hence, our framework and its results are influenced by the order of variants to be tested for one SPL version θ_i in time, e.g., the initial SPL version θ_0. In the literature, existing techniques for computing testing orders focus on the increase of the early fault detection [ATM+14; SSR14; DPC+17; EBA+11; JHF+12; LJC+14; PSS+16; HPP+14; DPL+16; BLL+14] by always selecting the most dissimilar variant as next variant to be tested. However, a dissimilarity-based testing order may influence our framework in a negative way by means of a decrease of the reuseability of test artifacts as well as test results between consecutively tested variants as their shared commonality may be rather small. To cope with this influence, we propose a prioritization technique which focuses on the similarity between variants to increase the exploitable reuse potentials by computing a reuse-optimized testing order. For the similarity examination, we incorporate the explicit knowledge about differences based on our delta-oriented test modeling formalism.

The remainder of this chapter is structured as follows. First, we introduce the incremental workflow for model-based regression testing of subsequent SPL versions under test in Sect. 5.1. Second, we propose the retest coverage criterion and the coverage-driven retest test selection in Sect. 5.2. Third, we describe the computation of reuse-optimized testing orders in Sect. 5.3. Fourth, we evaluate the efficiency as well as effectiveness of our model-based SPL regression testing framework and discuss the results in Sect. 5.4. Fifth, we discuss related work on model-based regression testing in the context of single-software systems and SPLs as well as related work regarding SPL product prioritization in Sect. 5.5. Finally, we conclude the chapter in Sect. 5.6.

5.1 A Workflow for Model-Based Regression Testing of Evolving Software Product Lines

In this section, we introduce the workflow of our model-based regression testing framework for evolving SPLs. The combination of model-based testing [UL06] and retest test selection as regression testing strategy [YH12] facilitates (1) the automatic generation of test cases based on state machine test models, i.e., the behavioral specification of a variant under test, and (2) the reduction of test cases to be (re-)executed for testing a variant or a version of a variant guided by applying change impact analysis. Furthermore, based on delta-oriented test modeling, we are able to exploit the reuse potential of test artifacts and test results during the regression testing of subsequent variants as well as SPL versions under test, where we incorporate the explicit knowledge about commonality and variability between variants and their versions. The workflow of our framework is based on the incremental SPL testing technique defined by Lochau et al. [LSK+12; LLL+14; Loc13], where variants are subsequently tested by reusing test artifacts and test results of already tested variants. In this thesis, we extend the existing workflow by integrating automated change impact analysis and coverage-driven retest test selection for regression testing of consecutively tested variants of a single SPL version. In addition, we adapt the workflow to take the evolution of SPLs into account facilitating the regression testing of subsequent SPL versions under test by reusing test artifacts and test results of preceding tested SPL versions.

Our framework starts the regression testing of an evolving SPL with its initial SPL version θ_0 as defined in the general workflow of our framework shown in Fig. 5.3. For this version, no test

artifacts and test results of preceding SPL versions exist to be reused such that we proceed slightly differently to the testing process of subsequent SPL versions to establish a test basis. We define the sub-workflow for regression testing of the initial version in the following section.

5.1.1 Regression Testing of the Initial SPL Version

The workflow of our framework for testing the initial SPL version θ_0 is shown in Fig. 5.1.

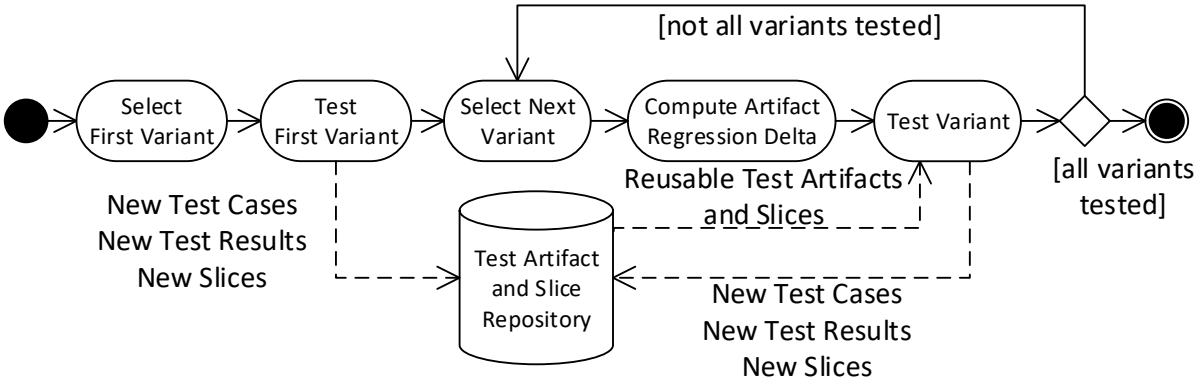


Figure 5.1: Workflow for Regression Testing of the Initial SPL Version [LNT+19]

We start the testing process by selecting the first variant $v_0 \in \mathbb{V}_{\theta_0}$ to be tested, where different selection scenarios exist for. Based on delta-oriented test modeling (cf. Chapt. 3), the core variant $v_{core} \in \mathbb{V}_{\theta_0}$ is a promising option to start from. Although the selection of v_{core} is a selection problem for itself as discussed in Sect. 3.2, we assume the core to denote a minimal variant comprising the most commonality, i.e., common behavior, between all variants of the initial SPL version θ_0 under test. This assumption is also shared by existing techniques for delta-oriented SPL testing [LLL+14; DSL+13; LLL+15; LNT+19; VBM15; DFG+17; SSR14]. Due to the commonality, the determined test artifacts have a high probability to be reused for subsequent variants under test. In contrast, the testing order and, hence, the first variant can be prescribed in advance based on the application of SPL prioritization techniques [HPP+14; ATL+16; LAT+17; PSS+16; LJC+14; DPC+13]. Existing techniques prioritize variants regarding their dissimilarity to each other incorporating problem space [HPP+14; ATL+16; PSS+16; LJC+14; SSR14] or solution-space artifacts [DPC+13; ALL+17]. Those techniques focus on the differences between variants to increase the test coverage, e.g., in terms of feature coverage, by always selecting the most dissimilar variant as next variant to be tested. However, a dissimilarity-based testing order may influence our framework by means of a decrease of the reuseability of test artifacts and test results between consecutively tested variants. To cope with this potential influence, we propose a prioritization technique which focuses on the similarity between variants w.r.t. their delta set differences. We introduce the delta-oriented similarity-based technique in Sect. 5.3. In addition, we investigate the potential impact of testing orders on the reuseability of test artifacts and test results during regression testing of the initial SPL version in Sect. 5.4. In this thesis, we exploit the testing order determined by our prioritization approach and select the first variant of the order as our starting point for regression testing of the initial SPL version θ_0. However, in general, our model-based SPL regression testing framework is independent from a certain testing order and we require an order to be given as input.

For the first variant v_0, we apply standard model-based testing (cf. Sect. 2.1.1) as no test artifacts and test results to be reused exist from previous testing processes. Hence, we first derive test goals TG_{v_0} from the respective state machine test model sm_{v_0}, where we use all-transition coverage [UL06] for the derivation in this thesis. Please note, in case the first variant $v_0 \neq v_{core}$ does not equal the core variant, we transform the core state machine test model $sm_{v_{core}}$ into the variant-specific test model sm_{v_0} of v_0 based on its delta set Δ_{v_0}. As second step, we generate for each test goal $tg \in TG_{v_0}$ a covering test case tc and collect it in the variant-specific test suite TS_{v_0} which is afterwards executed on the implementation of v_0 to obtain respective test results. A test case tc is represented by the state machine path ρ^{tc}_{sm} (cf. Def. 3.7) traversed during its execution and, hence, covers a test goal if its respective transition is contained in ρ^{tc}_{sm}. In this thesis, we abstract from the test-case generation process as our framework is independent from a certain test-case generator and assume test suites to be derivable by exploiting techniques from the literature [UPL12; AS12; SK13; ABC+13]. The third step is defined by applying slicing to build a basis for our incremental change impact analysis when stepping to the next variant under test as described in Sect. 4.1. As we use all-transition coverage to derive test goals guiding the test process, we also use test goals $tg \in TG$, i.e., transitions, as slicing criteria. Hence, we compute a state machine slice $slice^{sm_{v_0}}_{tg}$ for every transition test goal $tg \in TG_{v_0}$ of the test model sm_{v_0}. As the last step, we record all test artifacts $TA_{v_0} = (sm_{v_0}, TG_{v_0}, TS_{v_0})$, the initial slices $slice^{sm_{v_0}}$, and the test results of the first variant v_0 in a shared *test artifact repository*. We exploit the repository to facilitate the access to already created test artifacts, test results, and slices to be reused in upcoming testing processes of variants and versions of variants.

After we finished the test of the first variant v_0, we select the next variant v_1 to be tested from the given testing order. To step from the last tested variant v_i to the next one v_{i+1}, we need to compute the state machine regression delta $\Delta_{v_i,v_{i+1}}$ based on their respective delta sets Δ_{v_i} and $\Delta_{v_{i+1}}$. The regression delta $\Delta_{v_i,v_{i+1}}$ is required to adapt the test artifacts TA_{v_i} for the selected variant v_{i+1} and, therefore, for testing the next variant as described in the next section and shown in Fig. 5.2. We repeat the subsequent tasks of selecting and testing the next variant v_{i+1} based on its predecessor v_i until all variants $v \in \mathbb{V}_{\theta_0}$ of the initial SPL version θ_0 are tested.

Example 5.1: Model-Based Testing of First Variant

Assume the sample testing order of v_{core}, v_1, v_2, and v_3 is given as input for applying our model-based regression testing framework for testing the initial SPL version θ_0 of our running example. Hence, the core variant v_{core} is the first variant to be tested. The respective state machine test model $sm_{v_{core}}$ is defined in Ex. 3.1 and shown in Fig. 3.1. By appyling all-transition coverage to $sm_{v_{core}}$, we obtain the test goal set $TG_{v_{core}} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$ comprising eight transition test goals w.r.t. the eight transitions contained in the test model $sm_{v_{core}}$. To cover the transition test goals, we generate test cases and capture them in the test suite $TS_{v_{core}} = \{tc_{t_1}, tc_{t_2}, tc_{t_3}, tc_{t_4}, tc_{t_5}, tc_{t_6}, tc_{t_7}, tc_{t_8}\}$. For instance, the test case $tc_{t_7} = \rho^{tc_{t_7}}_{sm} = (\{t_3\}, \{t_1\}, \{t_2\}, \{t_6\}, \{t_7\})$ is represented by the state machine path defined in Ex. 3.1 and depicted in Fig. 3.2. Please note, we abstract from the test-case execution in our running example. In addition to those test artifacts $TA_{v_{core}} = (sm_{v_{core}}, TG_{v_{core}}, TS_{v_{core}})$, we apply our slicing technique to compute the initial slices $slice^{sm_{v_{core}}}$ used as basis for subsequent testing processes of variants under test of the initial SPL version θ_0. For example, we compute the slice

$slice_{t_5}^{sm_{v_{core}}}$ for transition t_5 described in Ex. 4.1 and shown in Fig. 4.1c. In the end, we record the test artifacts $TA_{v_{core}}$ as well as all initial slices $slice^{sm_{v_{core}}}$ in the shared test artifact repository.

5.1.2 Regression Testing of Variants

When stepping to the next variant v_{i+1} under test, we start its test process with the automated adaptation of the test artifacts TA_{v_i} of the previously tested variant v_i as shown in Fig. 5.2. For the automated adaptation, our framework restores TA_{v_i} from the shared test artifact repository and exploits the regression delta $\Delta_{v_i,v_{i+1}}$ capturing the differences between the respective state machine test models sm_{v_i} and $sm_{v_{i+1}}$ to perform three steps. First, we transform the state machine test model sm_{v_i} into $sm_{v_{i+1}}$ based on the application of the regression delta $sm_{v_{i+1}} = \mathsf{apply}_\delta(sm_{v_i}, \Delta_{v_i,v_{i+1}})$. Second, we reuse the applied change operations of the regression delta $\Delta_{v_i,v_{i+1}}$ and adapt the test goal set TG_{v_i} to obtain $TG_{v_{i+1}}$. For a remove operation $op = (\mathsf{rem}\, t) \in \Delta_{v_i,v_{i+1}}$ of a transition t, we remove the respective test goal tg_t from the test goal set. For an add operation $op = (\mathsf{add}\, t) \in \Delta_{v_i,v_{i+1}}$ of a transition t, we add a new test goal tg_t, accordingly. Third, we adapt the test suite TS_{v_i} to obtain $TS_{v_{i+1}}$, where we also take the change operations of the regression delta into account.

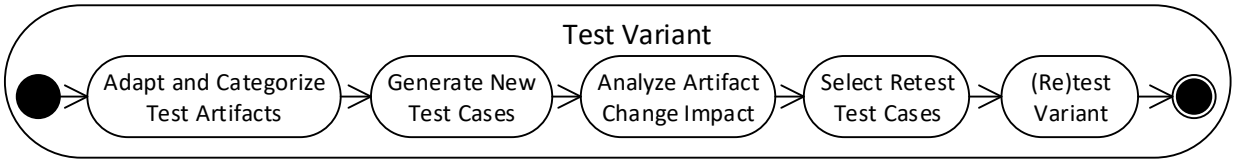


Figure 5.2: Workflow for Regression Testing of Variants and Versions of Variants [LNT+19]

For each test case tc of test suite TS_{v_i}, we examine if its representing state machine path ρ_{sm}^{tc} remains valid for the adapted test model $sm_{v_{i+1}}$, i.e., the sequence of transitions is not corrupted by a remove or modify operation of a transition captured in the regression delta $\Delta_{v_i,v_{i+1}}$. In case the state machine path ρ_{sm}^{tc} of a test case tc has become invalid, we remove tc from the test suite TS_{v_i} to be adapted. Afterwards, we check for each valid test case $tc \in TS_{v_i}$ if its simulated execution on test model $sm_{v_{i+1}}$ results in the same path ρ_{sm}^{tc} as for the previous test model sm_{v_i}. The simulation of a test-case execution on a test model can be performed similar to dynamic forward slicing [Bin98; ACH+13]. We iterate over the sequence of transitions of its path ρ_{sm}^{tc} determined on the previous test model sm_{v_i}, where we simulate the emergence of input events triggering the transitions of ρ_{sm}^{tc}. The simulation of test case tc on the adapted test model $sm_{v_{i+1}}$ results in one of three cases, namely (1) in the same path ρ_{sm}^{tc} as for sm_{v_i}, (2) in a slightly different path ρ_{sm}^{tc} as for sm_{v_i}, or (3) cannot be simulated anymore. In Case (1), tc remains in the test suite $TS_{v_{i+1}}$ and is reusable for the variant v_{i+1} to be tested. In contrast, for Case (2), where the simulation results in a slightly different state machine path ρ_{sm}^{tc}, the test case tc also remains in the test suite $TS_{v_{i+1}}$ for variant v_{i+1}, but we further map the new path ρ_{sm}^{tc} to tc such that we can exploit this information for the retest test selection defined in Sect. 5.2. A path ρ_{sm}^{tc} can be valid, yet differ between two state machine test models if ρ_{sm}^{tc} may traverse additional or even less transitions during its simulation. Such transitions do not belong to the main sequence of transitions that is traversed based on the simulation of the emergence of input events, but rather represents some kind of side-effect traversal due to the synchronization of transitions via internal events. In Case (3), the state machine regression delta

$\Delta_{v_i,v_{i+1}}$ adds a new transition that is now also traversed by the path ρ^{tc}_{sm} during the simulation. The additionally traversed transition blocks the remaining execution either (1) as the transition requires another input event to be triggered, or (2) the new transition is located on a higher state machine hierarchy level leaving the parent state of the subregions which contain the next transitions to be executed of the previous path ρ^{tc}_{sm}. We classify such test cases also as invalid and remove them from the test suite TS_{v_i} to be adapted, accordingly. In the end, the test suite adaptation results in the test suite $TS_{v_{i+1}}$ comprising solely valid and, hence, reusable test cases from the previous variant v_i.

As common in regression testing [YH12], we categorize test cases during the test suite adaptation in sets of *obsolete* $TS^O_{v_{i+1}}$, *reusable* $TS^R_{v_{i+1}}$, and *new* test cases $TS^N_{v_{i+1}}$. The category of obsolete test cases $TS^O_{v_{i+1}} \subseteq TS_{v_i}$ comprises all invalid test cases which are removed from the previous test suite TS_{v_i}. In contrast to standard regression testing techniques [YH12], where obsolete test cases are discarded, we solely omit them for the current variant v_{i+1} under test, but still store them in the shared test artifact repository as they may become reusable again for testing processes of subsequent variants. We categorize all test cases $tc \in TS_{v_i} \setminus TS^O_{v_{i+1}}$ that are also valid for the current variant v_{i+1} to be tested as reusable and collect them in the respective set $TS^R_{v_{i+1}}$. The category $TS^R_{v_{i+1}}$ of reusable test cases builds the basis for the application of our retest test selection which is defined in Sect. 5.2, whereas the reexecution of all reusable test cases is known as retest-all strategy [YH12]. The selection and retest of test cases is a crucial task of regression testing also known as the retest test selection problem [YH12] in order to revalidate that already tested behavior is not influenced other than intended. Hence, in the context of SPLs, we revalidate that the differences between subsequently tested variants that also exist in their respective implementations do not erroneously influence shared common behavior, e.g., by introducing unintended feature interactions.

After we finished the categorization of obsolete $TS^O_{v_{i+1}}$ and reusable test cases $TS^R_{v_{i+1}}$, we determine the set of uncovered test goals $tg \in TG_{v_{i+1}}$. When stepping to the next variant v_{i+1} to be tested, a test goal tg is uncovered because of (1) all its covering test cases $tc_{tg} \in TS_{v_i}$ contained in the test suite TS_{v_i} of the previous variant v_i are invalid and cannot be reused for variant v_{i+1}, i.e., $tc_{tg} \in TS^O_{v_{i+1}}$, or (2) the state machine element, i.e., transition, was not yet contained in a state machine test model sm_v of an already tested variant v such that there cannot exist a covering test case. For Case (1), we examine the set of test cases that are stored in the shared test artifact repository to identify test cases that are valid for variant v_{i+1} and also cover at least one of the uncovered test goals. To check if a test case tc stored in the repository is valid for v_{i+1}, we apply the same steps as for the identification of reusable test cases during the test suite adaptation, where its representing state machine path ρ^{tc}_{sm} is taken into account. All transitions that are contained in the path ρ^{tc}_{sm} also have to be contained in the test model $sm_{v_{i+1}}$ of v_{i+1} and the simulation of the test case on $sm_{v_{i+1}}$ results in a valid path as described above. If we find a valid test case tc in the repository, we add it to the test suite $TS_{v_{i+1}}$ and also categorize it as reusable $tc \in TS^R_{v_{i+1}}$. For Case (2) and also for those test goals which are still uncovered as we cannot find a covering and valid test case in the test artifact repository, we generate and add new test cases to $TS_{v_{i+1}}$ by applying techniques from the literature [UPL12; AS12; SK13; ABC+13]. Those test cases are further categorized as new in the set $TS^N_{v_{i+1}}$.

As result of the adaptation, we obtain a valid set of test artifacts $TA_{v_{i+1}} = (sm_{v_{i+1}}, TG_{v_{i+1}}, TS_{v_{i+1}})$ as well as a categorization of the test suite $TS_{v_{i+1}}$ in terms of obsolete $TS^O_{v_{i+1}}$, reusable $TS^R_{v_{i+1}}$, and new test cases $TS^N_{v_{i+1}}$ for testing variant v_{i+1}. Based on the state machine regression delta $\Delta_{v_i,v_{i+1}}$ and the adapted test model $sm_{v_{i+1}}$, we apply our incremental model slicing technique for change

impact analysis as described in Sect. 4.1 as next step of the workflow shown in Fig. 5.2. Each element addition and removal and, hence, the difference between the consecutively tested variants v_i and v_{i+1} may have an impact on shared common behavior that was already tested for v_i. Our change impact analysis detects such influences captured as slice differences indicating potentially affected behavior to be retested by selecting and retesting of reusable test cases.

For the slicing application, we first examine for each test goal $tg \in TG_{v_{i+1}}$ if a prior slice $slice_{tg}^{sm_{v_j}}$ exists in the shared test artifact repository computed for a previously tested variant v_j. By incorporating the last computed slice $slice_{tg}^{sm_{v_j}}$ for a test goal *tg*, we facilitate the identification of slice differences such that we are able to reason about retest decisions to be made for the current variant v_{i+1} under test. The last slice $slice_{tg}^{sm_{v_j}}$ was not necessarily computed for the previously tested variant v_i ($v_j \neq v_i$) if the respective test goal and, hence, its represented transition used as slicing criterion was not contained in the variant-specific test model sm_{v_i}. However, this slice selection strategy does not impede our change impact analysis for consecutively tested variants. In contrast, the selection strategy enhances the detection of changed execution dependencies of a slicing criterion, i.e., a transition, introduced based on the differences between the three variants v_j, v_i, and v_{i+1}. First, we take those differences into account existing between v_i and v_{i+1}, i.e., captured in the regression delta $\Delta_{v_i,v_{i+1}}$. Second, we also incorporate differences that solely exist between v_j and v_{i+1} such that we additionally detect their influences on the execution dependencies. Those differences already existed between v_j and v_i, where their impact was not completely apparent due to the absence of the slicing criterion not contained in the test model sm_{v_i} of variant v_i. Hence, the applied slice selection strategy facilitates a comprehensive impact analysis resulting in an improved retest test selection as described in Sect. 5.2. Furthermore, the selection strategy is influenced by the given testing order used for regression testing of the initial SPL version as depending on the order the result of the change impact analysis differs. We perform a controlled experiment regarding the influence of varying testing orders on our change impact analysis and retest test selection in Sect. 5.4.

An alternative slice selection strategy would be to determine for each test goal $tg \in TG_{v_{i+1}}$ its best fitting partner by means of the most similar variant w.r.t. the behavior represented by the transition used as test goal *tg*. The identification of the best fitting partner would facilitate that we detect less slice differences for *tg* and, hence, less retest potentials to reason about. However, the identification of the best fitting partner for every test goal *tg* on each variant v under test requires much more effort compared to selecting the last variant, especially, for those variants which are tested in the end of the test process of an SPL version under test. As the effort required for change impact analysis is a crucial cost factor influencing the efficiency of regression testing techniques according to Leung and White [LW91], the alternative selection strategy would impede our framework to be efficient. Hence, in this thesis, we focus on the selection strategy described above, where for a test goal $tg \in TG_{v_{i+1}}$, the last computed slice is taken into account for change impact analysis. By focusing solely on the last slice, we accept that we may determine more slice differences resulting in more retest decisions to be made compared to the strategy with the best fitting partner. Nevertheless, we argue that the effort which is required to retest the difference of identified retest potentials between both strategies by selecting and reexecuting respective test cases is lower than the additional effort required for the best fitting partner determination. Please note, our framework would also cope with the incorporation of the alternative selection strategy such that the determination of a best fitting partner is performed in advance of the incremental slice computation.

For a test goal *tg* for which we cannot find a previous slice $slice_{tg}^{sm_{v_j}}$ in the test artifact repository, we compute a new slice $slice_{tg}^{sm_{v_{i+1}}}$ and store it in the artifact repository for subsequent testing processes. This scenario occurs if the respective transition is contained in a variant-specific test model for the first time during regression testing of an SPL version. In contrast, if we find a previous slice $slice_{tg}^{sm_{v_j}}$ for *tg*, we restore the slice and exploit it for incrementally recomputing the slice $slice_{tg}^{sm_{v_{i+1}}}$ for the current variant v_{i+1} under test. As result of the incremental slice recomputation, we determine slice differences captured in a slice regression delta $\Delta_{v_j,v_{i+1}}^{slice_{tg}}$. The captured differences denote changed dependencies w.r.t. the slicing criterion, i.e., the transition test goal *tg* resulting from the applied test model changes. Changed dependencies may indicate potential sources of errors to be (re-)tested, e.g., due to unintended artifact or feature interactions. In case a slice regression delta $\Delta_{v_j,v_{i+1}}^{slice_{tg}} = \emptyset$ does not comprise any slice differences which implies that no changed dependencies exist, no retest potentials arise for the corresponding test goal *tg*.

We exploit the results of the impact analysis in our model-based regression testing framework to automatically reason about retest decisions of reusable test cases. For the reasoning process, we require a scale by means of an expressive criterion based on detected slice differences. Coverage criteria like all-transition coverage are a meaningful scale to control distinct aspects of the testing process [UL06], e.g., test-case generation [UPL12; AS12; SK13; ABC+13]. In this thesis, we adopt the concept of test coverage criteria to define a *retest coverage criterion* incorporating slice differences by deriving retest test goals in order to guide the retest test selection. The new coverage criterion as well as the retest test selection are defined in Sect. 5.2. Thus, each retest test goal has to be covered by at least one test case to ensure retest test coverage such that we select reusable test cases from $TS_{v_{i+1}}^{R}$ for a retest. The selected test cases are captured in a *retest test suite* $TS_{v_{i+1}}^{Re}$ to be executed in addition to the set of new test cases $TS_{v_{i+1}}^{N}$ for (re-)testing variant v_{i+1} under test. The (re-)testing of variant v_{i+1} represents the last step of the workflow depicted in Fig. 5.2 such that we store afterwards the variant-specific test artifacts $TA_{v_{i+1}}$ as well as slices $slice^{sm_{v_{i+1}}}$ in the shared test artifact repository for subsequent testing processes. Please note that the test suites may contain redundant test cases by means of retest as well as standard test goal coverage. Such redundancy can be further reduced to optimize the testing process by applying test-suite minimization techniques [YH12] which is out of the scope of this thesis. To continue the regression testing workflow of the initial SPL version $\mathbb{V}_0$ under test shown in Fig. 5.1, we select and test the next variant until no variants to be tested remain.

Example 5.2: Regression Testing of Next Variant

Consider Ex. 5.1 again, where we applied standard model-based testing to the first variant v_{core} of the sample testing order v_{core}, v_1, v_2, and v_3. The testing order defines the sequence of variant-specific testing processes for regression testing of the initial SPL version θ_0 of our running example. By stepping to the next variant v_1, we compute the regression delta $\Delta_{v_{core},v_1} = \Delta_{v_1}$ which is defined in Ex. 3.3 and start the test artifact adaptation of $TA_{v_{core}}$ to obtain TA_{v_1}. As first step, we adapt the state machine test model $sm_{v_1} = \mathsf{apply}_\delta(sm_{v_{core}}, \Delta_{v_{core},v_1})$ by applying the regression delta Δ_{v_{core},v_1} to the test model $sm_{v_{core}}$ of variant v_{core}. The resulting state machine test model sm_{v_1} is shown in Fig. 3.4b. As second step, we exploit the change operations captured in Δ_{v_{core},v_1} to adapt the test goal set TG_{v_1}. Based on the addition of transitions t_9, t_{10}, t_{11}, t_{12}, t_{13}, and t_{14}, we add respective transition test goals to the set TG_{v_1}.

In addition, we remove the test goal t_8 from TG_{v_1} as its represented transition is removed via Δ_{v_{core},v_1}. The resulting test goal set $TG_{v_1} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}\}$ of variant v_1 contains 13 transition test goals w.r.t. the 13 transitions comprised in sm_{v_1}.

As third step, we adapt the test suite $TS_{v_{core}}$ to obtain TS_{v_1} and further categorize test cases during the adaptation process. Based on the removal of test goal t_8, we also remove test case tc_{t_8} from the test suite and categorize it as obsolete. The other test cases $tc \in TS_{v_{core}} \setminus \{tc_{t_8}\}$ in the test suite $TS_{v_{core}}$ are not affected by the change operations captured in Δ_{v_{core},v_1}, i.e., their respective state machine paths ρ^{tc}_{sm} are not disrupted and still simulatable, and, therefore, remain in the test suite TS_{v_1} for variant v_1. Those test cases are categorized as reusable. Furthermore, we have to derive new test cases for the new test goals t_9, t_{10}, t_{11}, t_{12}, t_{13}, and t_{14} to ensure all-transition coverage. In the end, we obtain the test suite $TS_{v_1} = \{tc_{t_1}, tc_{t_2}, tc_{t_3}, tc_{t_4}, tc_{t_5}, tc_{t_6}, tc_{t_7}, tc_{t_9}, tc_{t_{10}}, tc_{t_{11}}, tc_{t_{12}}, tc_{t_{13}}, tc_{t_{14}}\}$ and also the categorization $TS^O_{v_1} = \{tc_{t_8}\}$ of obsolete test cases, $TS^R_{v_1} = \{tc_{t_1}, tc_{t_2}, tc_{t_3}, tc_{t_4}, tc_{t_5}, tc_{t_6}, tc_{t_7}\}$ of reusable test cases, and $TS^N_{v_1} = \{tc_{t_9}, tc_{t_{10}}, tc_{t_{11}}, tc_{t_{12}}, tc_{t_{13}}, tc_{t_{14}}\}$ of new test cases.

To identify changed influences, we apply incremental model slicing to analyze the change impact. Consider the slice $slice^{sm_{v_{core}}}_{t_5}$ for transition t_5 described in Ex. 4.1 and shown in Fig. 4.1c again. By recomputing the slice for v_1, we obtain the same slice and, hence, determine no slice differences indicating that no retest decisions are to be made for transition t_5. In contrast, the recomputation of the slice $slice^{sm_{v_{core}}}_{t_2}$ for transition t_2 depicted in Fig. 4.1b results in a different slice $slice^{sm_{v_1}}_{t_2}$ for variant v_1 as shown in Fig. 4.4, where the slice differences, i.e., additions of state machine elements, are marked by a $+$. The slice differences are captured as slice regression delta $\Delta^{slice_{t_2}}_{v_{core},v_1}$ used to derive retest test goals to be covered by selecting reusable test cases. We apply the impact analysis to all transitions of the test model sm_{v_1}, where for new transitions, we also compute new slices. As result we obtain the set $slice^{sm_{v_1}}$ of slices for variant v_1. As last step of the testing process of v_1, we record the test artifacts TA_{v_1} as well as all slices $slice^{sm_{v_1}}$ in the shared artifact repository and step to the next variant under test.

5.1.3 Regression Testing of Subsequent SPL Versions

After we finished the regression testing process of the initial SPL version θ_0 as described in the previous sections, we step to the next SPL version θ_1 under test and exploit the determined test artifacts, test results, and slices of the initial SPL version θ_0 stored in the shared test artifact repository for the test process of θ_1. The general workflow of our framework for regression testing of subsequent SPL versions θ_{i+1} based on its predecessor SPL version θ_i is shown in Fig. 5.3.

We start the testing process of the subsequent SPL version θ_{i+1} under test by applying the respective higher-order delta $\delta^H_{\theta_{i+1}}$ to the delta test model DM_{θ_i} to obtain the delta test model $DM_{\theta_{i+1}} = \mathsf{apply}_{\delta^H}(DM_{\theta_i}, \delta^H_{\theta_{i+1}})$ of version θ_{i+1}. In this context, we apply our variant set change impact analysis as described in Sect. 4.2, i.e., the reasoning about higher-order delta application, to examine the impact of the evolution step between the set of variants $\mathbb{V}_{\theta_i}$ of the previous SPL version θ_i and $\mathbb{V}_{\theta_{i+1}}$ of the current SPL version θ_{i+1} to be tested. The result of the impact analysis is a categorization by means of added, removed, modified, and unchanged variants. We exploit this information to guide the regression testing process of the SPL version θ_{i+1} under test as described in the following

Figure 5.3: Workflow for Model-Based SPL Regression Testing [LNT+19]

paragraphs. We focus on modified as well as new variants, whereas unchanged variants are skipped as we already (re)tested those variants in the previous SPL version θ_i. Unchanged variants are not affected by the evolution step from SPL version θ_i to version θ_{i+1}, i.e., they are equal in the respective variant sets $\mathbb{V}_{\theta_i}$ as well as $\mathbb{V}_{\theta_{i+1}}$ and, hence, no retest has to be applied for SPL version θ_{i+1}.

Regression Testing of Modified Variant Versions. Based on the categorization, we first test all modified variant versions as shown in Fig. 5.3. Modified variants $v_j \in \mathbb{V}_{\theta_{i+1}}$ are tested based on their previous versions $v_j \in \mathbb{V}_{\theta_i}$, where a certain testing order is not required. We exploit the mapping between variant versions specified by the mapping of variant-tree paths $\Delta_{\rho_{VT}}^{\theta_{i+1}}$ of the current SPL version θ_{i+1} to the variant-tree paths $\Delta_{\rho_{VT}}^{\theta_i}$ of the preceding version θ_i determined during the incremental delta set derivation described in Sect. 4.2.3. Based on this mapping, we perform the following steps for regression testing of modified variant versions just as defined in the previous Sect. 5.1.2:

1. Compute the state machine regression delta $\Delta_{v_j^{\theta_i}, v_j^{\theta_{i+1}}}$ between both versions by incorporating their version-specific delta sets $\Delta_{v_j^{\theta_i}}$ and $\Delta_{v_j^{\theta_{i+1}}}$
2. Transform the state machine test model $sm_{v_j^{\theta_i}}$ into $sm_{v_j^{\theta_{i+1}}}$ by applying $\Delta_{v_j^{\theta_i}, v_j^{\theta_{i+1}}}$
3. Adapt the test goal set $TG_{v_j^{\theta_i}}$ to obtain $TG_{v_j^{\theta_{i+1}}}$ based on the examination of $\Delta_{v_j^{\theta_i}, v_j^{\theta_{i+1}}}$
4. Adapt and categorize the test suite $TS_{v_j^{\theta_i}}$ to get $TS_{v_j^{\theta_{i+1}}}$ as well as $TS^O_{v_j^{\theta_{i+1}}}$, $TS^R_{v_j^{\theta_{i+1}}}$, and $TS^N_{v_j^{\theta_{i+1}}}$
5. Apply incremental model slicing to identify changed dependencies to be retested
6. Select test cases for (re)testing the modified version $v_j^{\theta_{i+1}}$

After we finished the test of a modified variant, we select the next one until all modified variants are tested based on their respective original versions of the preceding SPL version θ_i under test. Please note, in case no modified variant is detected by our variant set change impact analysis, we directly continue with the testing process of new variants.

Regression Testing of New Variants. As shown in Fig. 5.3, we test all new variants of SPL version θ_{i+1} after we finished or skipped the regression testing of modified variant versions. In contrast to this previous testing process, we require a testing order to be given for regression testing of the new variants. As already described in Sect. 5.1.1, our framework is, in general, independent from a specific testing order. Therefore, existing prioritization techniques [HPP+14; AKT+16a; LAT+17] as well as our delta-oriented similarity-based technique (cf. Sect. 5.3) are applicable. However, for this testing step, we exploit the structure of the incrementally created variant tree $VT_{\theta_{i+1}}$ for the specification of a testing order. In a variant tree VT, similar variants v and v' and, therefore, their respective variant tree paths ρ_{VT}^{v} and $\rho_{VT}^{v'}$ are located next to each other which follows from the variant tree definition (cf. Def. 4.10). By starting from the most left leaf tree node of a variant tree, we add each variant v to the testing order which is represented by the variant tree path ρ_{VT}^{v} derivable from the current leaf node under consideration and further categorized as new, i.e., $\rho_{VT}^{v} \in \Delta_{\rho_{VT}}^{new}$ holds.

Furthermore, for testing the first new variant of the determined testing order, we exploit the testing processes for unchanged as well as modified variants. We again use the variant tree to determine the most similar variant compared to the first variant to be tested which was already tested for the current SPL version θ_{i+1} under test. Hence, the first variant is tested based on its most similar partner and the remaining new variants are consecutively tested following the determined testing order. For testing consecutive variants, we apply the same steps as described above such that (1) the

state machine regression delta is computed, (2) the test artifacts are adapted, (3) the change impact is analyzed, and (4) test cases are selected and executed for (re)testing the subsequent variant under test. Please note, in case no new variant is identified by our variant set change impact analysis, the test process for the current SPL version θ_{i+1} under test is finished and we step to the next SPL version to be tested after evolution occurs as depicted in Fig. 5.3.

Example 5.3: Regression Testing of Subsequent SPL Version

Consider again Ex. 3.5 for the higher-order delta definition and application of $\delta^H_{\theta_1}$ as well as Ex. 4.8 for the reasoning about its application based on the incremental delta set derivation. We obtained a categorization of variants, where we identified v_4 as new variant and v'_{core}, v'_1, v'_2, and v'_3 as modified variants. In this evolution step, no unchanged variants were detected.

Based on the mapping between variants of the previous SPL version θ_0 and the current SPL version θ_1 (cf. Ex. 4.8), we test each modified variant w.r.t. its original version. For instance, the modified variant $v_1 \in \mathbb{V}_{\theta_1}$ is tested based on its original variant version $v_1 \in \mathbb{V}_{\theta_0}$. First, we compute the regression delta $\Delta_{v_1^{\theta_0}, v_1^{\theta_1}}$ (cf. Ex. 4.4) and apply it to the state machine test model $sm_{v_1}^{\theta_0}$ to obtain the test model $sm_{v_1}^{\theta_1}$ shown in Fig. 4.5a. Second, we adapt the test goal set such that we obtain $TG_{v_1^{\theta_1}} = TG_{v_1^{\theta_0}} \cup \{t_{19}, t_{20}, t_{21}, t_{22}, t_{23}\}$, where five transition test goals are added w.r.t. the addition of the respective transitions via $\Delta_{v_1^{\theta_0}, v_1^{\theta_1}}$. Third, we update and categorize the test suite such that we get $TS_{v_1^{\theta_1}} = TS_{v_1^{\theta_0}} \cup \{tc_{t_{19}}, tc_{t_{20}}, tc_{t_{21}}, tc_{t_{22}}, tc_{t_{23}}\}$, where five new test cases are derived to ensure all-transition coverage. The categorization is specified by $TS^O_{v_1^{\theta_1}} = \emptyset$, $TS^R_{v_1^{\theta_1}} = TS_{v_1^{\theta_0}}$, and $TS^N_{v_1^{\theta_1}} = \{tc_{t_{19}}, tc_{t_{20}}, tc_{t_{21}}, tc_{t_{22}}, tc_{t_{23}}\}$. Fourth, we apply incremental model slicing to analyze the change impact. For example, we recompute the slice $slice_{t_2}^{sm_{v_1}^{\theta_1}}$ for transition t_2 depicted in Fig. 4.5b, where the difference, i.e., the addition of transition t_{19} is marked by a $+$. The slice difference is captured as slice regression delta $\Delta^{slice_{t_2}}_{v_1^{\theta_0}, v_1^{\theta_1}}$ which is used to derive a respective retest test goal to be covered by selecting reusable test cases. After we finished the test of the modified variant v_1, we step to the next one until all modified variants are tested based on their original versions from the already tested SPL version θ_0.

For testing the new variant v_4, we determine the most similar variant from the current SPL version θ_1 under test as starting point for its testing process. As variant v_4 is categorized as new, it cannot be tested based on a respective original version from SPL version θ_0 and further cannot be tested based on a preceding new variant of SPL version θ_1 as it is the first and only new variant to be tested. Consider the variant tree VT_{θ_1} in Fig 4.11a as well as the derivable variant tree paths described in Ex. 4.8. The most similar variant for v_4 is v'_3 as they differ solely in the selection of delta δ_6 as can be seen in the following variant tree paths:

$$\rho^{v'_3}_{VT} = ((n^R_{\delta_6}, n^L_{\delta_5}, n^L_{\delta_4}, n^R_{\delta_3}, n^L_{\delta_2}, n^L_{\delta'_1}),$$
$$\lambda_{\prec}(n_{\delta_6}) = \lambda_{\prec}(n_{\delta_3}) = R; \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta'_1}) = L)$$
$$\rho^{v_4}_{VT} = ((n^L_{\delta_6}, n^L_{\delta_5}, n^L_{\delta_4}, n^R_{\delta_3}, n^L_{\delta_2}, n^L_{\delta'_1}),$$
$$\lambda_{\prec}(n_{\delta_3}) = R; \lambda_{\prec}(n_{\delta_6}) = \lambda_{\prec}(n_{\delta_5}) = \lambda_{\prec}(n_{\delta_4}) = \lambda_{\prec}(n_{\delta_2}) = \lambda_{\prec}(n_{\delta'_1}) = L)$$

Based on the selection of variant v'_3, we test the new variant v_4 by performing the same steps as presented above.

5.2 Retest Test Selection for Variants and Versions of Variants

In this section, we describe the retest test selection integrated in our model-based regression testing framework for reducing the number of test cases to be executed based on the results of incremental model slicing applied as change impact analysis. We select reusable test cases for a retest to validate that changes do not unintentionally influence already tested behavior when stepping to a subsequent variant under test also known as retest test selection problem [YH12]. For reasoning about retest decisions, we require a scale by means of an expressive criterion based on detected slice differences. Coverage criteria like all-transition coverage are a meaningful scale to control distinct aspects of the testing process [UL06], e.g., test-case generation [UPL12; AS12; SK13; ABC+13]. Therefore, we adopt the concept of test coverage criteria to define a *retest coverage criterion* that belongs to the class of change-based coverage criteria [FWT+11] and incorporates slice differences by deriving retest test goals in order to guide the retest test selection. The retest coverage criterion as well as the coverage-based retest test selection are described in the following sections.

5.2.1 Retest Coverage Criterion

When stepping from the last tested variant v_i to the next variant or version of a variant v_j to be tested, we apply our incremental slicing technique for change impact analysis. As result, we obtain, for each transition test goal *tg* used as slicing criterion, a slice regression delta $\Delta_{v_i,v_j}^{slice_{tg}}$ capturing the slice differences, i.e., changes to the execution dependencies of the transition represented by the test goal *tg*. An empty slice regression delta $\Delta_{v_i,v_j}^{slice_{tg}} = \emptyset$ indicates that no retest potentials exist for the test goal *tg*, whereas slice differences $|\Delta_{v_i,v_j}^{slice_{tg}}| > 0$ imply retest potentials, i.e., already tested behavior to be revalidated during regression testing [GB08; YH12]. We select reusable test cases to be reexecuted such that an identified slice difference, e.g., added element, as well as the test goal *tg* used as slicing criterion are traversed. Based on the traversal, we revalidate that the already tested behavior represented by the transition used as test goal *tg* and slicing criterion still behaves as expected and is not erroneously influenced by the slice difference, i.e., changed execution dependency. Hence, to guide the retest test selection of our model-based regression testing framework, we take the detected slice differences into account to derive retest test goals. A *retest test goal* is defined by a pair of state machine test model elements, where we consider two cases for the derivation such that the first element *elem* is either (1) a state or a transition added to the recomputed slice $slice_{tg}^{sm_{v_j}}$ via an respective add operation $(\text{add } elem) = op^{Slice} \in \Delta_{v_i,v_j}^{slice_{tg}}$, or (2) a source or target state of a transition removed during the slice computation via a remove operation $(\text{rem } t) = op^{Slice} \in \Delta_{v_i,v_j}^{slice_{tg}}$. The second element is given by the test goal *tg* for which slice differences are detected and captured in $\Delta_{v_i,v_j}^{slice_{tg}}$.

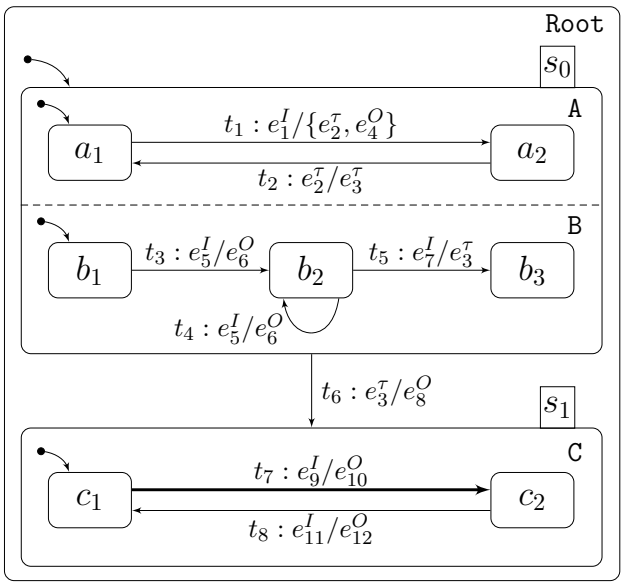
Definition 5.1: Retest Test Goal

A *retest test goal* $rtg = (elem, tg)$ is defined as tuple, where

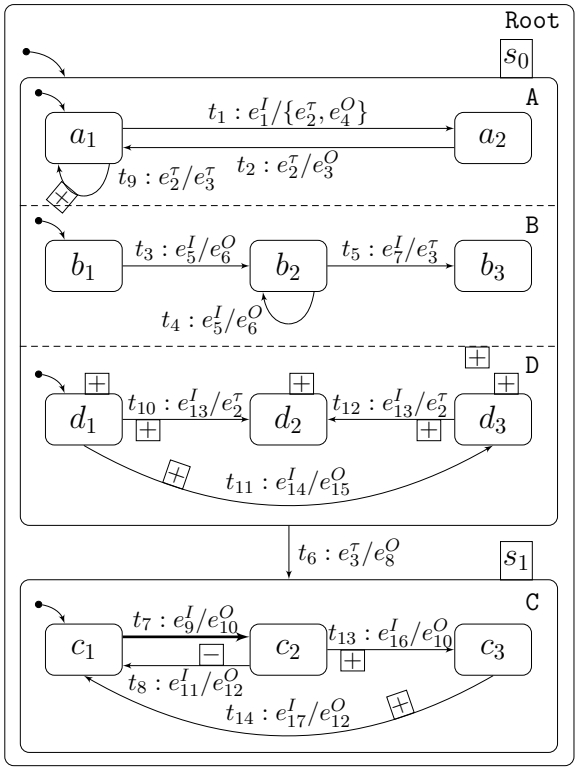
- $elem \in S_{R_{sm_{v_j}}} \cup T_{R_{sm_{v_j}}}$ is a *state machine element* affected by a slice difference, and
- $tg \in TG_{v_j}$ is a *test goal* for which slice differences are detected.

For the retest test goal derivation, we consider Case (1) to (re-)validate newly introduced dependencies between the added element *elem* and the slicing criterion *tg*. In contrast, in Case (2), we

validate that removed behavior represented by a removed transition does not still exist in the implementation of variant v_j, and that new dependencies introduced due to the removal are implemented as expected. We collect all derived retest test goals in a respective test goal set $TG^{R}_{v_j}$ used to select reusable test cases such that we ensure 100% retest coverage as described in the next section.



(a) Sample State Machine Slice $slice^{sm_{v_{core}}}_{t_7}$ for Transition t_7 in State Machine Test Model $sm_{v_{core}}$ of Core Variant v_{core}



(b) Recomputed State Machine Slice $slice^{sm_{v_1}}_{t_7}$ for Transition t_7 in State Machine Test Model sm_{v_1} of Variant v_1

Figure 5.4: Original and Recomputed Slices $slice^{sm}_{t_7}$ for Transition t_7 Including Slice Differences

Example 5.4: Retest Test Goal Derivation

Consider Ex. 5.2 again, where we described the incremental testing step from variant v_{core} to the subsequent variant v_1. By applying our incremental model slicing technique for change impact analysis, we obtain the same slice $slice^{sm_{v_1}}_{t_5}$ recomputed for transition t_5 for v_1 as for v_{core}. Hence, the slice regression delta $\Delta^{slice_{t_5}}_{v_{core},v_1} = \emptyset$ is empty and we do not derive retest test goals for transition t_5. In contrast, for transition t_7, the slice regression delta $\Delta^{slice_{t_7}}_{v_{core},v_1} = \{\text{add } t_9, \text{add } d_1, \text{add } d_2, \text{add } d_3, \text{add } t_{10}, \text{add } t_{11}, \text{add } t_{12}, \text{rem } t_8, \text{add } c_3, \text{add } t_{13}, \text{add } t_{14}\}$ comprises ten additions and one removal indicating retest potentials for transition t_7. The slice $slice^{sm_{v_{core}}}_{t_7}$ for the core variant v_{core} is shown in Fig. 5.4a and the recomputed slice $slice^{sm_{v_1}}_{t_7}$ for variant v_1 is depicted in Fig. 5.4b, where the slice differences also captured in $\Delta^{slice_{t_7}}_{v_{core},v_1}$ are marked by a $+$ for additions and by a $-$ for the removal. We derive 12 retest test goals for retesting behavior w.r.t. transition t_7 collected in the retest test goal set $TG^{R}_{v_1}$, namely $rtg_{t_9,t_7} = (t_9, t_7)$, rtg_{d_1,t_7}, rtg_{d_2,t_7}, rtg_{d_3,t_7}, rtg_{t_{10},t_7}, rtg_{t_{11},t_7}, rtg_{t_{12},t_7}, rtg_{c_1,t_7}, rtg_{c_2,t_7}, rtg_{c_3,t_7}, rtg_{t_{13},t_7}, and rtg_{t_{14},t_7}. The retest test

goals rtg_{c_1,t_7} as well as rtg_{c_2,t_7} are derived based on the removal of transition t_8, i.e., state c_2 represents the source state and c_1 the target state of t_8, whereas the remaining retest test goals are based on the additions of state machine elements to the slice of t_7.

5.2.2 Retest Test Selection and Generation

To cover a retest test goal $rtg = (elem, tg) \in TG^R_{v_j}$ derived for retesting a variant v_j, a test case tc has to traverse both elements *elem* and *tg* via its state machine path ρ^{tc}_{sm}, where $\rho^{tc}_{sm} = (\ldots, T_{t_{elem}}, \ldots, T_{t_{tg}})$ holds. By $t_{elem} \in T_{t_{elem}}$, we refer (1) directly to the transition represented by *elem* or (2) to the transition that has *elem* as target state if *elem* is a state. In addition, $t_{tg} \in T_{t_{tg}}$ denotes the transition for which the test goal was derived and the slice was incrementally recomputed. This way of covering a retest test goal is beneficial as we ensure that the influence of a changed execution dependency w.r.t. the transition t_{tg} is directly revalidated via the reexecution of a test case. An alternative way of covering a retest test goal would be to select all reusable test cases that traverse the transition t_{tg}, but may have no relation or dependency to the changed influence denoted by the slice difference. By selecting all reusable test cases that traverse t_{tg}, we further obtain a larger number of test cases to be retested comprising more test cases which are unnecessarily reexecuted compared to the number of test cases our framework selects by focusing on the changed execution dependency and their influence on t_{tg}. Hence, a test case tc covering a retest test goal $rtg = (elem, tg)$ denotes a representative execution of variant v_j to be retested validating that no unexpected behavior is implemented based on new or changed dependencies/interactions between both state machine elements *elem* and *tg*.

Similar to model-based coverage criteria [UL06], for each retest test goal $rtg = (elem, tg) \in TG^R_{v_j}$ at least one covering test case tc has to exist. We select reusable test cases $tc \in TS^R_{v_j}$ for a retest on the current variant v_j. Please note, retest test goals are potentially also covered by new test cases $tc \in TS^N_{v_j}$, which are generated for transitions that are newly added via the state machine regression delta Δ_{v_i,v_j} to ensure all-transition coverage, as their state machine path traverses both state machine elements of retest test goal *rtg*. However, some retest goals may not be covered by the current set $TS^R_{v_j}$ of reusable test cases as their respective state machine paths do not traverse both elements or do not traverse the elements in the correct order. For such cases, we again apply test-case generation to derive specific *retest test cases* for covering the remaining retest test goals. According to Bates and Horwitz [BH93] as well as Rothermel and Harrold [RH94], this is reasonable to ensure that already tested, yet change-affected behavior is revalidated. Those retest test cases are not generated solely for the retest process of the current variant v_j, but are further recorded in the shared test artifact repository such that they are reusable for subsequently tested variants or version of variants.

We collect the selected reusable test cases and the newly generated retest test cases in a respective *retest test suite* $TS^{Re}_{v_j}$. Both, the retest test suite $TS^{Re}_{v_j}$ and the set of new test cases $TS^N_{v_j}$ are executed for (re)testing variant v_j. Afterwards, we record all test artifacts and test results in the shared test artifact repository and continue with the workflow of our model-based regression testing framework by stepping to the next variant or version of a variant to be tested (cf. Sect. 5.1). The executed test suites may contain several test cases that provide the same (re)test goal coverage indicating further redundancy to be reduced. Hence, there is still optimization potential, e.g., by applying test-suite minimization techniques [YH12], which is, however, out of the scope of this thesis.

Example 5.5: Retest Test Selection and Generation

Consider Ex. 5.4 again, where we derived 12 retest test goals for retesting already tested behavior w.r.t. transition t_7. To cover those retest test goals, different coverage scenarios arise, namely (1) we select reusable test cases to ensure coverage, (2) new test cases cover retest test goals, and (3) we explicitly generate retest test cases. For instance, the test case tc_{t_7} defined in Ex. 5.1 and categorized as reusable test case in Ex. 5.2 is selected for reexecution as it covers the retest test goal rtg_{c_1,t_7}. In addition, the new test case $tc_{t_{13}}$ generated to ensure all-transition coverage and represented by the state machine path $\rho_{sm}^{tc_{t_{13}}} = (\{t_{10}\}, \{t_9\}, \{t_6\}, \{t_7\}, \{t_{13}\})$ covers the retest test goals rtg_{t_{10},t_7}, rtg_{t_9,t_7}, and rtg_{c_1,t_7}. For covering the retest test goals rtg_{t_{11},t_7} as well as rtg_{t_{12},t_7}, we generate a retest test case tc_{t_{12},t_7} which is denoted by the state machine path $\rho_{sm}^{tc_{t_{12},t_7}} = (\{t_{11}\}, \{t_{12}\}, \{t_9\}, \{t_6\}, \{t_7\})$. Hence, the newly generated retest test case rtg_{t_{11},t_7} further covers the retest test goals rtg_{t_9,t_7} and rtg_{c_1,t_7}. The remaining retest test goals of the retest test goal set $TG_{v_1}^R$ are covered accordingly w.r.t. the three different coverage scenarios.

5.3 Optimized Testing Orders for Model-Based Regression Testing of Software Product Lines

To reduce the overall testing effort for testing evolving SPLs, our model-based regression testing framework exploits the reuse potential between consecutively tested variants $v \in \mathbb{V}_\theta$ of an SPL version θ as described in Sect. 5.1. The reuse potential exists based on the commonality shared by subsequent variants under test. Our framework exploits the shared commonality and focuses on the differences between variants to perform retest test selection guided by the results of the application of our slicing-based change impact analysis when stepping to the next variant to be tested. In general, our regression testing framework does not depend on a certain testing order of variants such that our framework is applicable to, e.g., total as well as partial orders of variants of an SPL version under test which can be provided by varying prioritization strategies. However, as already discussed in Chapt. 4, the results of the impact analysis are influenced by the order in which variants are analyzed. The more similar subsequently analyzed variants are, the greater is the potential reduction of retest decisions to be made as similar variants imply less retest test goals to be retested. To increase the exploitable reuse potential of test artifacts as well as test results during regression testing of an SPL version, a respective optimized testing order is desirable.

Existing techniques applicable to determine testing orders, prioritize variants of an SPL version regarding their dissimilarity to each other [HPP+14; ATL+16; PSS+16; LJC+14; SSR14; DPC+13; ALL+17]. Hence, those techniques focus on the differences between variants to increase the test coverage, e.g., in terms of feature coverage, by always selecting the most dissimilar variant as next variant to be tested. Based on such a coverage-driven testing order, the early fault detection and, therefore, the testing effectiveness is increased. However, a dissimilarity-based testing order may influence our framework by means of a decrease of the reuseability of test artifacts and test results between consecutively tested variants as their shared commonality may be rather small. To cope with this potential influence, we propose a prioritization technique which focuses on the similarity between variants to increase the testing efficiency. We focus on the testing efficiency to reduce the

overall test effort by determining a difference-optimized testing order of variants under test following a similarity-based prioritization strategy. For the similarity examination, we incorporate the explicit knowledge about differences captured by regression deltas based on our delta-oriented test modeling formalism defined in Chapt. 3. Based on the delta information, we are able to determine a testing order by specifying an optimization problem such that the overall differences between subsequently tested variants gets minimized. We determine a sequential, i.e., total order of variants similar to the existing prioritization techniques [HPP+14; ATL+16; PSS+16; LJC+14; SSR14; DPC+13; ALL+17] which is further supported by the workflow of our regression testing framework described in Sect. 5.1. Please note, as efficiency and effectiveness are contradicting testing objectives, future prioritization techniques should also focus on an optimized trade-off to facilitate a combination of efficient retest test selection and effective fault detection.

To find a similarity-driven testing order, we encode the respective optimization problem as well-known *traveling salesperson problem* (TSP) [Rei94; ABC+11]. Based on this encoding, we are able to identify a sequence of variants to be tested, where the total number of differences between subsequent variants is minimized. As solving a given TSP and, therefore, finding an optimal solution is NP-complete [Rei94; ABC+11], we adopt existing heuristics to find an approximately optimal testing order supporting model-based regression testing of an SPL version under test.

In the following, we describe the encoding of our optimization problem as TSP and, afterwards, we describe the adoption of graph-based heuristics for solving our optimization problem. An experiment regarding the influence of varying testing orders on our framework is performed in Sect. 5.4.

5.3.1 Problem Encoding as Traveling Salesperson Problem

For the encoding of the optimized testing order problem as TSP, we define a *weighted variant graph* comprising a set of variant nodes, a finite set of edges connecting variant nodes, and a weighting function. Each variant node represents a respective variant $v \in \mathbb{V}$ of all variants of an SPL version for which the prioritization technique is to be applied. In addition, each edge denotes the potential step between two variants $v_i, v_j \in \mathbb{V}$ during regression testing of the SPL under consideration, where the weighting function further specifies a weight for each edge to facilitate the reasoning about distances between variants.

Definition 5.2: Weighted Variant Graph

A *weighted variant graph* $G_{\mathbb{V}} = (N_{\mathbb{V}}, \rightarrow_{\omega}, \omega)$ is defined as triple, where

- $N_{\mathbb{V}} = \{n_{v_0}, \ldots, n_{v_n}\}$ is a finite set of *variant nodes*,
- $\rightarrow_{\omega} \subseteq N_{\mathbb{V}} \times N_{\mathbb{V}}$ is a finite *graph edge relation*, and
- $\omega : \rightarrow_{\omega} \rightarrow \mathbb{N}$ is a *weighting function*.

For the weighting function ω, different instantiations are possible to provide weights for edges of a variant graph $G_{\mathbb{V}}$. In the literature [HPP+14; AKT+16a], promising candidates for ω are the Hamming or Jaccard distance measurements incorporating the feature configurations of variants. As our model-based regression testing framework focuses on the differences between state machine test models of subsequently tested variants, we abstract from the comparison of feature configurations and exploit the explicit specification of differences between variants $v_i, v_j \in \mathbb{V}$ for the weight derivation based on our delta-oriented test-modeling formalism (cf. Chapt. 3). In addition, the dif-

ferences between variant-specific test models are captured as state machine regression deltas Δ_{v_i,v_j} such that we do not require the application of the Hamming or Jaccard distance measurements for the comparison. We use the sum of change operations contained in a regression delta Δ_{v_i,v_j} to define a weight for an edge $n_{v_i} \rightarrow_\omega n_{v_j}$ connecting the variant nodes $n_{v_i}, n_{v_j} \in N_\mathbb{V}$ such that

$$\omega(n_{v_i} \rightarrow_\omega n_{v_j}) = |\Delta_{v_i,v_j}| = |\{op_1, \ldots, op_m\}|$$

holds. For the summation of the change operations, we incorporate the addition, removal, and modification of a state machine element equally. Thus, each operation $op \in \Delta_{v_i,v_j}$ has the same impact to the model and is counted with 1. This interpretation facilitates the definition of our weighted variant graph to be *undirected*. Please note, in case we incorporate change operations differently w.r.t. their change type, i.e., addition, modification, or removal, the definition of our weighted variant graph results in a *directed* graph. In a directed variant graph, each edge would have a concrete direction as the regression deltas Δ_{v_i,v_j} and Δ_{v_j,v_i} used for the weight computation contains different change operations (cf. Def. 3.13) and, therefore, will result in different weights.

For a successful encoding as TSP, a variant graph $G_\mathbb{V}$ needs to satisfy three additional properties [ABC+11; Rei94]. First, the graph $G_\mathbb{V}$ has to be *complete*, i.e., for every pair of variants $v_i, v_j \in \mathbb{V}$, there exists exactly one edge $n_{v_i} \rightarrow_\omega n_{v_j} \in \rightarrow_\omega$ connecting the respective variant nodes $n_{v_i}, n_{v_j} \in N_\mathbb{V}$. This property is ensured due to the equal interpretation of the distinct change operation types resulting in an undirected graph. Second, the graph $G_\mathbb{V}$ has to be *connected*, i.e., for every pair of variants $v_i, v_j \in \mathbb{V}$, a path between the respective variant nodes $n_{v_i}, n_{v_j} \in N_\mathbb{V}$ within $G_\mathbb{V}$ exists. A *variant graph path* between variant nodes n_{v_i} and n_{v_j} of a variant graph $G_\mathbb{V}$ is defined as a sequence of variant graph edges starting in node n_{v_i} and ending in node n_{v_j}.

Definition 5.3: Variant Graph Path

Let $\rightarrow_\omega^*$ be the set of all variant graph paths of a weighted variant graph $G_\mathbb{V}$. A *variant graph path* $\rho_{G_\mathbb{V}} = (\rightarrow_\omega^1, \ldots, \rightarrow_\omega^k) \in \rightarrow_\omega^*$ of length k of a variant graph $G_\mathbb{V}$ is a sequence of variant graph edges such that the following holds

- $\rightarrow_\omega^1 = n_{v_i} \rightarrow_\omega n_{v'}$, i.e., the path starts in variant node n_{v_i} ,
- $\rightarrow_\omega^k = n_{v''} \rightarrow_\omega n_{v_j}$, i.e., the path ends in variant node n_{v_j}, and
- $\forall \rightarrow_\omega^r = n_{v_{r-1}} \rightarrow_\omega n_{v_r}, 1 < r < k$: $\rightarrow_\omega^{r+1} = n_{v_r} \rightarrow_\omega n_{v_{r+1}}$, i.e., the target variant node of an edge is the source variant node of the subsequent edge.

The second property directly follows from the completeness of the graph $G_\mathbb{V}$. As last property, the graph $G_\mathbb{V}$ has to fulfill the *triangle inequality*, i.e., for all distinct variants $v_i, v_j, v_m \in \mathbb{V}$ holds

$$\omega(v_i \rightarrow_\omega v_j) \leq \omega(v_i \rightarrow_\omega v_m) + \omega(v_m \rightarrow_\omega v_j).$$

The fulfillment of the triangle inequality follows from the definition of state machine regression deltas (cf. Def. 3.13), where every difference between two variants is captured as change operation, i.e.,

$$|\Delta_{v_i,v_j}| \leq |\Delta_{v_i,v_m}| + |\Delta_{v_m,v_j}|.$$

For instance, assume variant v_m to be the core variant. The regression delta $\Delta_{v_i,v_m} = (\Delta_{v_i})^{-1}$ represents the inverted delta set for variant v_i and the regression delta $\Delta_{v_m,v_j} = \Delta_{v_j}$ denotes the delta set

to transform the core into variant v_j. As the computation of the regression delta Δ_{v_i,v_j} is dependent on the variant-specific delta sets Δ_{v_i} and Δ_{v_j}, the delta Δ_{v_i,v_j} comprises at most the same change operations as both delta sets combined. Therefore, the triangle inequality is fulfilled by the variant graph $G_{\mathbb{V}}$ as the regression delta derivation is applicable between arbitrary variants of an SPL.

Based on the definition of a weighted variant graph and its properties, we instantiate a (symmetric) TSP [Rei94] to find a tour $\rho_{G_{\mathbb{V}}}^{min}$ within a given graph $G_{\mathbb{V}}$ such that the total number of differences between subsequent variants is minimized and every variant node is visited once. A *tour* denoting a TSP solution is specified as a variant graph path $\rho_{G_{\mathbb{V}}}$ starting and ending in the same variant node. The total number of differences is computed as the sum of edge weights of a tour $\rho_{G_{\mathbb{V}}}$ and, therefore, by the number of change operations captured in state machine regression deltas such that

$$\omega(\rho_{G_{\mathbb{V}}}) = \sum_{n_{v_i} \rightarrow_\omega n_{v_j} \in \rho_{G_{\mathbb{V}}}} \omega(n_{v_i} \rightarrow_\omega n_{v_j})$$

holds. Our instantiation of the TSP is defined as follows:

Given: Weighted Variant Graph $G_{\mathbb{V}}$
Find: Variant Graph Tour $\rho_{G_{\mathbb{V}}}^{min}$ such that the following holds

$$\neg\exists \rho'_{G_{\mathbb{V}}} : \omega(\rho'_{G_{\mathbb{V}}}) < \omega(\rho_{G_{\mathbb{V}}}^{min})$$

By solving the instantiated TSP, we obtain an optimized tour $\rho_{G_{\mathbb{V}}}^{min}$ within the given graph $G_{\mathbb{V}}$. However, for the determination of an optimized testing order, we solely require a path that traverses every variant node of the variant graph once and has different start as well as end nodes. To solve the TSP and to determine solely an optimized graph path, we extend the input weighted variant graph by adding a dummy node D which is connected to all other variant nodes $n_v \in N_{\mathbb{V}}$ via respective edges with an assigned weight of zero. Based on this extension, the TSP can be solved resulting in an optimized tour starting and ending in the dummy node D. To derive the respective variant graph path from the tour, we remove the dummy node and its two connecting edges from the tour. The resulting path represents an optimal testing order, where the total number of differences between variants is minimized to increase the reuse potential between subsequent variants under test.

However, solving a TSP is NP-complete [Rei94; ABC+11], where the effort to find an optimal solution is dependent on the number of variants to be analyzed. To cope with larger variant sets defined by the complete set of variants of an SPL under test or by a computed variant set sample [VAT+18], we adopt existing heuristics for solving a given TSP by approximating from the optimal solution [Rei94] as described in the next section.

Table 5.1: Symbol Summary of Testing Order Problem Encoding

Symbol	Description
$G_{\mathbb{V}}$	Weighted variant graph
$n_v; N_{\mathbb{V}}$	Variant node; Finite set of variant nodes
$\rightarrow_\omega$	Variant graph edge relation
ω	Weighting funtion
$\rho_{G_{\mathbb{V}}}; \rightarrow_\omega^*$	Variant graph path; Finite set of variant graph paths

We summarize the list of symbols used for the encoding definition of the testing order problem as TSP in Tab. 5.1. To recapitulate, a weighted variant graph $G_{\mathbb{V}} = (N_{\mathbb{V}}, \rightarrow_{\omega}, \omega)$ comprises a set of variant nodes $N_{\mathbb{V}}$, where for every variant $v \in \mathbb{V}$ of the SPL under consideration, a respective node $n_v \in N_{\mathbb{V}}$ exists. A variant graph further contains the edge relation $\rightarrow_{\omega}$ such that for every pair of variants $v_i, v_j \in \mathbb{V}$, an edge $n_{v_i} \rightarrow_{\omega} n_{v_j}$ connects their variant nodes $n_{v_i}, n_{v_j} \in N_{\mathbb{V}}$ denoting the potential step between the two variants during SPL regression testing. Based on the weighting function ω, each edge $n_{v_i} \rightarrow_{\omega} n_{v_j} \in \rightarrow_{\omega}$ gets an weight assigned $\omega(n_{v_i} \rightarrow_{\omega} n_{v_j}) = |\Delta_{v_i,v_j}|$, where the number of change operations captured in the regression delta Δ_{v_i,v_j} is taken into account to specify the value. Based on the definition of a weighted variant graph $G_{\mathbb{V}}$ and its properties, we are able to instantiate a symmetric TSP to find an optimized testing order, where the total number of differences between subsequent variants to be tested gets minimized. Such an optimized testing order focuses on the similarity between subsequently variants to be tested resulting in a potential reduction of retest decisions to be made by means of less derivable retest test goals to be retested. By solving the instantiated TSP, we obtain a variant graph path $\rho_{G_{\mathbb{V}}} \in \rightarrow_{\omega}^{*}$ of the set of all graph paths specified as sequence of variant graph edges denoting an optimized testing order.

Example 5.6: Weighted Variant Graph and Optimal Testing Order

Consider the sample weighted variant graph $G_{\mathbb{V}_{\theta_0}}$ for the initial SPL version θ_0 of our running example shown in Fig. 5.5a. Based on the variant set $\mathbb{V}_{\theta_0}$ as well as the state machine regression deltas Δ_{v_i,v_j} between variants $v_i, v_j \in \mathbb{V}_{\theta_0}$, the variant graph $G_{\mathbb{V}_{\theta_0}}$ is defined by

- $N_{\mathbb{V}_{\theta_0}} = \{v_{core}, v_1, v_2, v_3\}$,
- $\rightarrow_{\omega} = \{v_{core} \rightarrow_{\omega} v_1, v_{core} \rightarrow_{\omega} v_2, v_{core} \rightarrow_{\omega} v_3, v_1 \rightarrow_{\omega} v_2, v_1 \rightarrow_{\omega} v_3, v_2 \rightarrow_{\omega} v_3\}$, and
- $\omega : \omega(v_{core} \rightarrow_{\omega} v_1) = 6; \omega(v_{core} \rightarrow_{\omega} v_2) = 2; \omega(v_{core} \rightarrow_{\omega} v_3) = 10; \omega(v_1 \rightarrow_{\omega} v_2) = 8; \omega(v_1 \rightarrow_{\omega} v_3) = 10; \omega(v_2 \rightarrow_{\omega} v_3) = 8.$

For instance, the edge $v_{core} \rightarrow_{\omega} v_1$ between the variants v_{core} and v_1 has the assigned weight

$$\omega(v_{core} \rightarrow_{\omega} v_1) = |\Delta_{v_{core},v_1}| = |\Delta_{v_1}| = 6.$$

To instantiate a TSP based on this graph $G_{\mathbb{V}_{\theta_0}}$, we extend it by the dummy node D as depicted in Fig. 5.5b. By solving the respective TSP, we obtain the optimal tour

$$\rho_{G_{\mathbb{V}}}^{min} = (D \rightarrow_{\omega} v_3, v_3 \rightarrow_{\omega} v_2, v_2 \rightarrow_{\omega} v_{core}, v_{core} \rightarrow_{\omega} v_1, v_1 \rightarrow_{\omega} D)$$

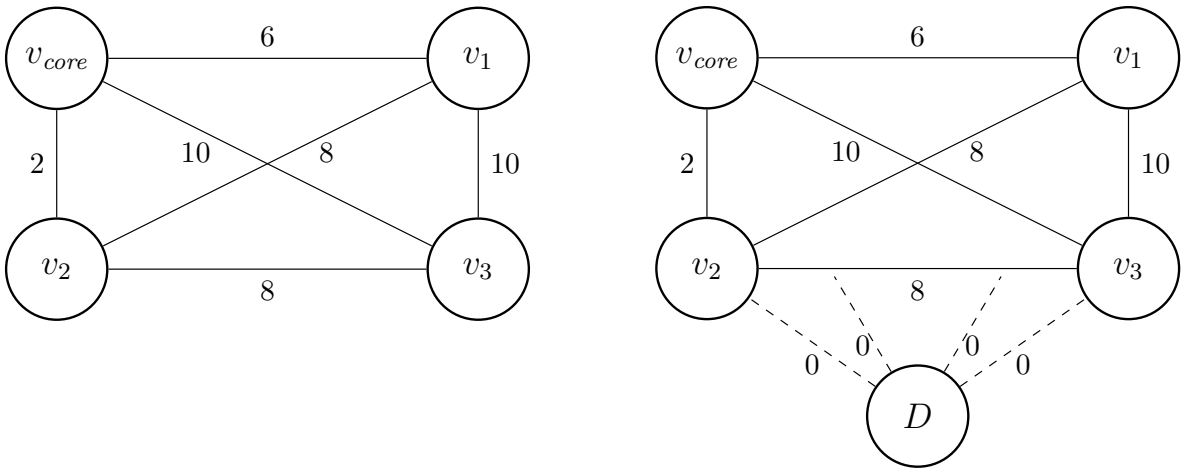
starting and ending in the dummy node with $\omega(\rho_{G_{\mathbb{V}}}^{min}) = 16$ as total number of differences. We remove the dummy node to determine the optimal path

$$\rho_{G_{\mathbb{V}}}^{min} = (v_3 \rightarrow_{\omega} v_2, v_2 \rightarrow_{\omega} v_{core}, v_{core} \rightarrow_{\omega} v_1)$$

resulting in the optimized testing order v_3, v_2, v_{core}, and v_1.

5.3.2 Application of Heuristics for Solving Traveling Salesperson Problems

For solving a TSP, several heuristics exist to approximate the optimal solution [Rei94; ABC+11]. In this thesis, we adopt two types of heuristics, namely greedy-based and insertion heuristics, to find a testing order, where the total number of differences between subsequent variants gets minimized



(a) Sample Weighted Variant Graph $G_{\mathbb{V}}$

(b) Extended Weighted Variant Graph $G_{\mathbb{V}}$

Figure 5.5: Sample Weighted Variant Graph for SPL Version θ_0 with Dummy Node Extension

to exploit an increased reuse potential during SPL regression testing. We describe the adoption in the following paragraphs.

Greedy-Based Heuristics. The first heuristic we adopt is the *nearest neighbor heuristic* which is a *greedy* algorithm [Rei94]. Based on a given weighted variant graph $G_{\mathbb{V}}$ and a certain variant node n_v used as starting point, we always select and add the nearest neighbor of the last visited variant node. The nearest neighbor denotes the most similar variant by means of minimal edge weight and, therefore, minimal differences. As a weighted variant graph fulfills the triangle inequality, we find the next variant by examining the adjacent variant nodes and their connecting weighted edges. In contrast to the original nearest neighbor heuristic [Rei94], where an optimized tour is found within the input weighted graph, our adoption determines an optimized path $\rho^{min}_{G_{\mathbb{V}}}$ such that the heuristic is finished after we integrate the edge connected to the last not yet visited variant node into the resulting path $\rho^{min}_{G_{\mathbb{V}}}$. For the selection of the first variant node n_v to start from, in general, any node can be chosen. However, based on our delta-oriented test-modeling formalism, we start with the respective variant node $n_{v_{core}}$ of the core variant v_{core} as it builds the basis for all defined state machine deltas specifying the differences between variants used for the assignment of edge weights.

In summary, we perform the following steps to find an optimized path $\rho^{min}_{G_{\mathbb{V}}}$ using the nearest neighbor heuristic:

1. Select the variant node $n_{v_{core}}$ of the core variant v_{core} as starting node
2. Find the edge $n_{v_i} \rightarrow_\omega n_{v_j}$ with minimal weight $\omega(n_{v_i} \rightarrow_\omega n_{v_j})$ w.r.t. the last visited variant node n_{v_i}, where the node n_{v_j} was not yet visited in the path $\rho^{min}_{G_{\mathbb{V}}}$ to be found
3. Add the edge $n_{v_i} \rightarrow_\omega n_{v_j}$ to the end of path $\rho^{min}_{G_{\mathbb{V}}} = (n_{v_{core}} \rightarrow_\omega n_{v'}, \ldots, n_{v''} \rightarrow_\omega n_{v_i})$
4. Repeat Step 2 and 3 until all variant nodes are visited via the path $\rho^{min}_{G_{\mathbb{V}}}$

The application of the nearest neighbor heuristic terminates resulting in an approximation of the optimal solution as the number of variant nodes to be visited during the computation is finite and based on the completeness property satisfied by a variant graph $G_{\mathbb{V}}$, where every variant node is reachable from every other variant node in $G_{\mathbb{V}}$. However, the heuristic has the potential drawback that in the first steps, we add very similar variants to the testing order represented by the optimized

graph path $\rho^{min}_{G_{\mathbb{V}}}$, but we may add very dissimilar ones during the last integration steps as we have no insights about the remaining variants to be added such that the difference between the optimal solution and the computed approximation increases.

Therefore, we improve the heuristic by incorporating a *look up* during the determination of the optimized variant graph path $\rho^{min}_{G_{\mathbb{V}}}$. For the selection of the nearest neighbor, we examine whether the next variant node to be visited via a minimal weighted edge can be found from the start or end of the already computed path $\rho^{min}_{G_{\mathbb{V}}}$. Based on this look up, we find a more suitable variant node to be added enhancing the resulting approximation. For the extension of the nearest neighbor heuristic, we split up its Step 2 into three new steps 2*a*, 2*b*, and 2*c* and further update its Step 3. In the new Step 2*a*, we determine the most similar adjacent variant node compared to the variant node representing the starting point of the current path. In contrast, we identify the most similar adjacent variant node w.r.t. the last variant node of the current path in Step 2*b*. Based on this information, we select in Step 2*c*, the edge connecting one of the recommended variant nodes which has the minimal weight to either the first or last node of the current path $\rho^{min}_{G_{\mathbb{V}}}$. In addition, Step 3 is updated such that the selected edge is added to the begin or end of the path $\rho^{min}_{G_{\mathbb{V}}}$, respectively. Please note, due to the potential integration of a selected edge at the beginning of $\rho^{min}_{G_{\mathbb{V}}}$, the core variant may not be the first variant of the resulting testing order to start from.

In summary, we perform the following steps to find an optimized variant graph path $\rho^{min}_{G_{\mathbb{V}}}$ based on the extended nearest neighbor heuristic:

1. Select the variant node $n_{v_{core}}$ of the core variant v_{core} as starting node

2a. Find the edge $n_{v'} \rightarrow_{\omega} n_{v_j}$ with minimal weight $\omega(n_{v'} \rightarrow_{\omega} n_{v_j})$ w.r.t. the current start variant node n_{v_j}, where the node $n_{v'}$ was not yet visited in the path $\rho^{min}_{G_{\mathbb{V}}}$ to be found

2b. Find the edge $n_{v_i} \rightarrow_{\omega} n_v$ with minimal weight $\omega(n_{v_i} \rightarrow_{\omega} n_v)$ w.r.t. the last visited variant node n_{v_i}, where the node n_v was not yet visited in the path $\rho^{min}_{G_{\mathbb{V}}}$ to be found

2c. Select the edge which has the minimal weight $min(\omega(n_{v_i} \rightarrow_{\omega} n_v), \omega(n_{v'} \rightarrow_{\omega} n_{v_j}))$ or select the edge $n_{v_i} \rightarrow_{\omega} n_v$ in case both weights are equal

3. Add the selected edge $n_{v'} \rightarrow_{\omega} n_{v_j}$ or $n_{v_i} \rightarrow_{\omega} n_v$ to the respective end of the path $\rho^{min}_{G_{\mathbb{V}}} = (n_{v_j} \rightarrow_{\omega} n_{v''}, \ldots, n_{v'''} \rightarrow_{\omega} n_{v_i})$

4. Repeat Step 2*a*, 2*b*, 2*c* and 3 until all variant nodes are visited via the path $\rho^{min}_{G_{\mathbb{V}}}$

The application of the extended heuristic also terminates as the set of variant nodes $N_{\mathbb{V}}$ to be visited via the path to be found is finite and based on the completeness property of a variant graph $G_{\mathbb{V}}$. The extended nearest neighbor heuristic provides a better approximation of the optimal solution. However, the extension does not completely prevent from the integration of dissimilar variants in the last steps of the heuristic. To cope with this issue, we exploit existing insertion heuristics for solving a TSP [Rei94], where an optimized path $\rho^{min}_{G_{\mathbb{V}}}$ to be computed is not solely extended at one of its ends, but also in between to facilitate an improved approximation of the optimal solution as described in the next section.

Example 5.7: Application of Greedy-Based Heuristics

Consider the weighted variant graph $G_{\mathbb{V}_{\theta_0}}$ for the initial SPL version θ_0 defined in Ex. 5.6 and shown in Fig. 5.5a again. By applying the nearest neighbor heuristic to this graph starting in the variant node v_{core} of the core variant, we have to find the first minimal weighted edge to

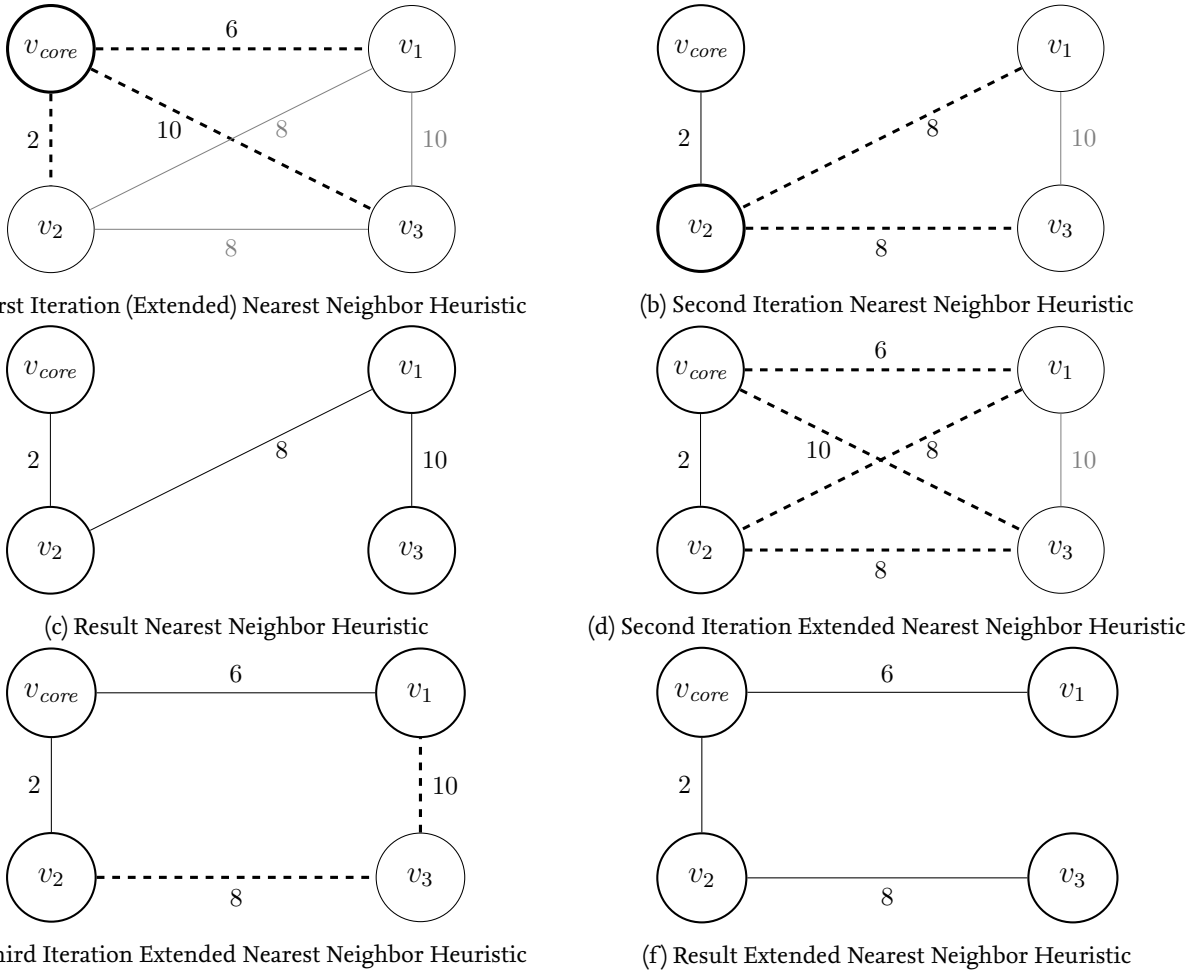
be integrated in the optimized path $\rho^{min}_{G_{\mathbb{V}}}$. As depicted in Fig. 5.6a, we have to select one of the highlighted dashed edges which has the minimal weight. Hence, in the first iteration, we add the edge $v_{core} \rightarrow_{\omega} v_2$ to the path $\rho^{min}_{G_{\mathbb{V}}} = (v_{core} \rightarrow_{\omega} v_2)$ as this edge has the minimal weight of $\omega(v_{core} \rightarrow_{\omega} v_2) = 2$. In the second iteration shown in Fig. 5.6b, we have to choose between the highlighted dashed edges $v_2 \rightarrow_{\omega} v_1$ and $v_2 \rightarrow_{\omega} v_3$ to find the next edge to be integrated in $\rho^{min}_{G_{\mathbb{V}}}$. Both edges have the same minimal weight of 8 such that we select one of the edges randomly. Assume we select $v_2 \rightarrow_{\omega} v_1$ to be integrated as next edge into $\rho^{min}_{G_{\mathbb{V}}} = (v_{core} \rightarrow_{\omega} v_2, v_2 \rightarrow_{\omega} v_1)$. As depicted in Fig. 5.6c, there is solely one edge left to be integrated. Therefore, $v_1 \rightarrow_{\omega} v_3$ is added to $\rho^{min}_{G_{\mathbb{V}}}$. The optimized path $\rho^{min}_{G_{\mathbb{V}}} = (v_{core} \rightarrow_{\omega} v_2, v_2 \rightarrow_{\omega} v_1, v_1 \rightarrow_{\omega} v_3)$ represents the result of the application of the nearest neighbor heuristic, i.e., the testing order is defined as v_{core}, v_2, v_1, and v_3 with an total minimal number of differences of 20.

By applying the extended nearest neighbor heuristic to this graph starting in the variant node v_{core} of the core variant, we, again, have to find the first minimal weighted edge from the three highlighted dashed edges shown in Fig. 5.6a. Just as for the original nearest neighbor heuristic described above, we add the edge $v_{core} \rightarrow_{\omega} v_2$ to the path $\rho^{min}_{G_{\mathbb{V}}}$. Afterwards, we identify the potential candidate edges to choose from for the next path extension from boths ends of the current path $\rho^{min}_{G_{\mathbb{V}}}$, where the options for v_{core} and for v_2 are highlighted as dashed lines as depicted in Fig. 5.6d. Based on the minimal weight of 6, we add the edge $v_{core} \rightarrow_{\omega} v_1$ to the beginning of the current optimized path such that $\rho^{min}_{G_{\mathbb{V}}} = (v_1 \rightarrow_{\omega} v_{core}, v_{core} \rightarrow_{\omega} v_2)$ holds. In the third iteration of the extended nearest neighbor heuristic shown in Fig. 5.6e, we have to select either the edge $v_1 \rightarrow_{\omega} v_3$ or $v_2 \rightarrow_{\omega} v_3$ as last edge to be integrated. As $v_2 \rightarrow_{\omega} v_3$ has the minimal weight of $\omega(v_2 \rightarrow_{\omega} v_3) = 8$, we add the edge to the end of $\rho^{min}_{G_{\mathbb{V}}}$. The resulting optimized path $\rho^{min}_{G_{\mathbb{V}}} = (v_1 \rightarrow_{\omega} v_{core}, v_{core} \rightarrow_{\omega} v_2, v_2 \rightarrow_{\omega} v_3)$ is depicted in Fig. 5.6f representing the testing order v_1, v_{core}, v_2, and v_3 with an total minimal number of differences of 16.

We can see that the extended nearest neighbor heuristic provides compared to the original nearest neighbor heuristic a better approximation of the optimal solution which was determined in Ex. 5.6. For our running example, the extended heuristic even achieves the same result as the optimal solution, where the testing order is solely specified inversely.

Insertion Heuristics. Besides the nearest neighbor heuristics, we apply insertion heuristics [Rei94] to solve an instantiation of the optimized testing order problem encoded as TSP. Insertion heuristics facilitate the computation of an optimized tour $\rho^{min}_{G_{\mathbb{V}}}$. Therefore, we have to exploit the dummy node extension of a weighted variant graph $G_{\mathbb{V}}$ to derive an approximately optimal path representing an optimized testing order w.r.t. a minimized total number of differences between subsequent variants under test. In this thesis, we apply (1) the **near**est **in**sertion (NEARIN), and (2) the **far**thest **in**sertion (FARIN) heuristic. Those heuristics provide good results when solving a given TSP as discussed in the literature [Rei94; ABC+11].

Both heuristics differ in the selection of the next variant node to be visited by integrating respective connecting edges in the tour $\rho^{min}_{G_{\mathbb{V}}}$ to be computed. Similar to the greedy-based heuristics, we exploit the core variant v_{core} and, therefore, its variant node $n_{v_{core}}$ as starting point to build an initial tour comprising the edge $D \rightarrow_{\omega} n_{v_{core}}$ connecting the dummy node D and the core variant



(a) First Iteration (Extended) Nearest Neighbor Heuristic

(b) Second Iteration Nearest Neighbor Heuristic

(c) Result Nearest Neighbor Heuristic

(d) Second Iteration Extended Nearest Neighbor Heuristic

(e) Third Iteration Extended Nearest Neighbor Heuristic

(f) Result Extended Nearest Neighbor Heuristic

Figure 5.6: Sample Application of Greedy-Based Heuristics

node $n_{v_{core}}$. For NEARIN, we always determine the variant node which is connected to a variant node of the current tour via an edge with the minimal weight. Please note, that for the selection of the variant node, we do not incorporate the edges which connect the dummy node with the other variant nodes as this would bias the tour computation due to the zero weights. For FARIN, we determine the variant node which is connected to a variant node of the current tour via an edge with the maximal weight. In contrast to the greedy-based heuristics, where we extend the optimized path to be found either at the beginning or the end of the path, we determine for the selected variant node its best fitting position in the tour. The best fitting position is defined such that the connection of the selected variant node with two variant nodes of the current tour $\rho^{min}_{G_V}$ result in the minimal increase of the overall number of differences to be minimized. As shown in Fig. 5.7, we remove the old connecting red edge between those identified variant nodes and add the green egdes that connect the new variant node with the variant nodes of the current tour. As last step of both heuristics, we remove the dummy node and its connecting edges in order to derive an optimized path from the resulting tour $\rho^{min}_{G_V}$ representing the optimized testing order.

In summary, we perform the following steps to find an optimized variant graph path $\rho^{min}_{G_V}$ based on the nearest as well as farthest insertion heuristic:

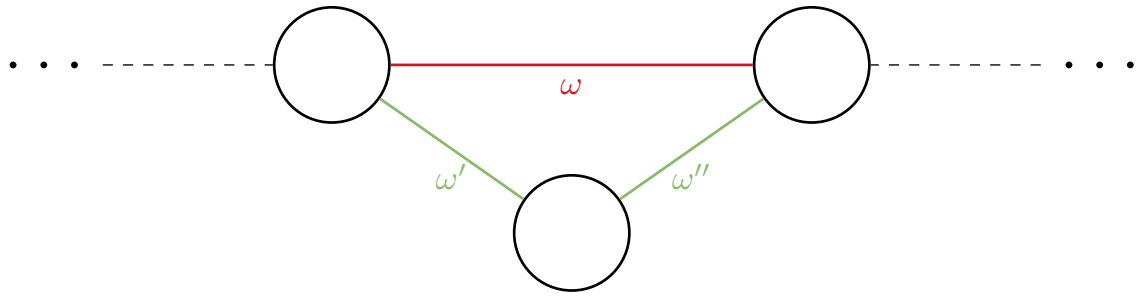


Figure 5.7: Extension of an Optimized Tour

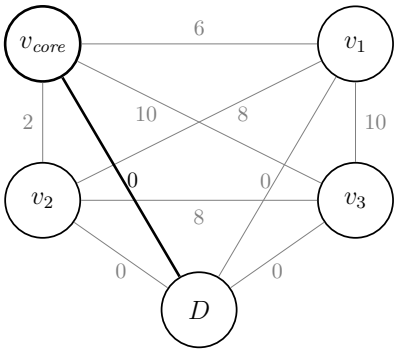
1. Build initial tour $\rho^{min}_{G_{\mathbb{V}}} = (D \rightarrow_{\omega} v_{core})$ with the edge $D \rightarrow_{\omega} v_{core}$
2. Find the edge with the minimal (NEARIN)/maximal (FARIN) weight connected to a variant node visited in the current tour $\rho^{min}_{G_{\mathbb{V}}}$ and select its target variant node
3. Identify the best fitting position to integrate the selected variant node in $\rho^{min}_{G_{\mathbb{V}}}$
4. Remove obsolete edge from $\rho^{min}_{G_{\mathbb{V}}}$ and add edges connecting the selected variant node with the identified best fitting position
5. Repeat Step 2 to 4 until all variant nodes are visited via the tour $\rho^{min}_{G_{\mathbb{V}}}$
6. Remove the edges connecting the dummy node D with other variant nodes from the tour $\rho^{min}_{G_{\mathbb{V}}}$ to derive an optimized path

Similar to the nearest neighbor heuristics, the application of both insertion heuristics terminate based on the finite set of variant nodes $N_{\mathbb{V}}$ to be visited via the tour to be found and the completeness property of a variant graph $G_{\mathbb{V}}$. The resulting approximation of the optimal solution determined by the insertion heuristics NEARIN and FARIN are exploited by our model-based regression testing framework for testing the initial SPL version. A testing order derivable from an optimized path $\rho^{min}_{G_{\mathbb{V}}}$ facilitates a reduction of the differences between subsequently tested variants such that our slicing-based change impact analysis and the retest test selection benefits from an increased reuse potential during SPL regression testing as described in the next section.

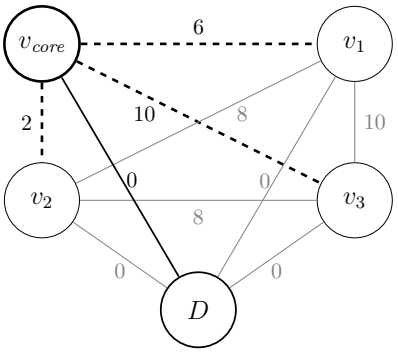
Example 5.8: Application of Insertion Heuristics

Consider the variant graph $G_{\mathbb{V}_{\theta_0}}$ for the initial SPL version θ_0 defined in Ex. 5.6 and shown in Fig. 5.5a again. By applying the insertion heuristics, we first have to build an initial tour $\rho^{min}_{G_{\mathbb{V}}} = (D \rightarrow_{\omega} v_{core})$ comprising the edge between the dummy node D and the variant node v_{core} of the core variant as depicted in Fig. 5.8a. Afterwards, we have to find the first variant node to be incorporated in the optimized tour $\rho^{min}_{G_{\mathbb{V}}}$. Depending on the applied insertion heuristic, the selection of the first node differs, where the weights of adjacent edges are taking into account. In Fig. 5.8b, the three potential candidates are highlighted by dashed edges.

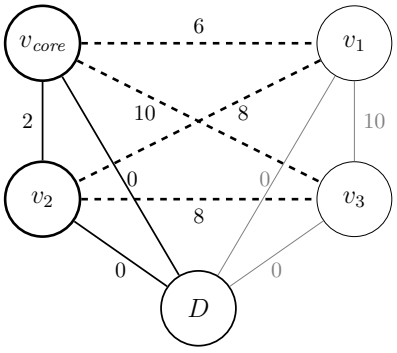
In the context of the NEARIN heuristic, variant node v_2 is selected and the tour is reconnected such that $\rho^{min}_{G_{\mathbb{V}}} = (D \rightarrow_{\omega} v_{core}, v_{core} \rightarrow_{\omega} v_2, v_2 \rightarrow_{\omega} D)$ holds as depicted in Fig. 5.8c. For this intermediate tour, we have to find the next variant node to be integrated such that we determine the adjacent edge with the minimal weight. As depicted in Fig. 5.8d, variant node v_1 is selected based on its connecting edge $v_{core} \rightarrow_{\omega} v_1$ that has an assigned weight of $\omega(v_{core} \rightarrow_{\omega} v_1) = 6$ and the tour is reconnected, accordingly. In the last iteration, the



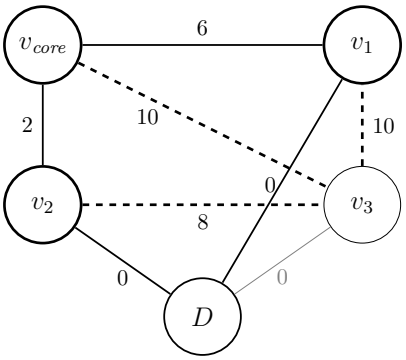
(a) Initial Tour Insertion Heuristics



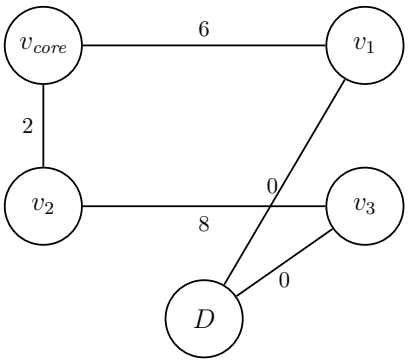
(b) First Iteration Insertion Heuristics



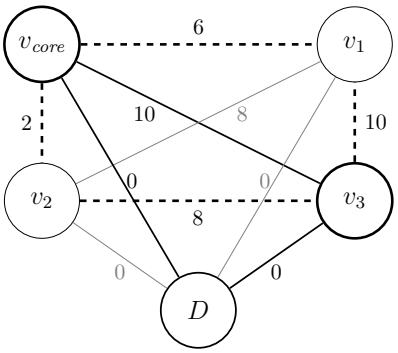
(c) Second Iteration NEARIN Heuristic



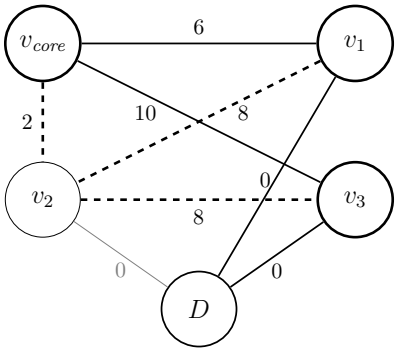
(d) Third Iteration NEARIN Heuristic



(e) Result Insertion Heuristics



(f) Second Iteration FARIN Heuristic



(g) Third Iteration FARIN Heuristic

Figure 5.8: Sample Application of Insertion Heuristics

variant node v_3 is integrated as shown in Fig. 5.8e. By removing the dummy node and its two connecting edges, we obtain the optimized path $\rho^{min}_{G_V} = (v_1 \rightarrow_\omega v_{core}, v_{core} \rightarrow_\omega v_2, v_2 \rightarrow_\omega v_3)$ representing the testing order v_1, v_{core}, v_2, and v_3 with an total weight of 16.

In the context of the FARIN heuristic, variant node v_3 is selected and the tour is reconnected such that $\rho^{min}_{G_V} = (D \rightarrow_\omega v_{core}, v_{core} \rightarrow_\omega v_3, v_3 \rightarrow_\omega D)$ holds as depicted in Fig. 5.8f. For this intermediate tour, we have to find the next variant node to be integrated such that we determine the adjacent edge with the maximal weight. As depicted in Fig. 5.8g, variant node v_1 is selected based on its connecting edge $v_3 \rightarrow_\omega v_1$ that has an assigned weight of $\omega(v_3 \rightarrow_\omega v_1) = 10$ and the tour is reconnected, accordingly. In the last iteration, the variant node v_2 is integrated as shown in Fig. 5.8e. By removing the dummy node and its respective edges, we obtain the same optimized path $\rho^{min}_{G_V} = (v_1 \rightarrow_\omega v_{core}, v_{core} \rightarrow_\omega v_2, v_2 \rightarrow_\omega v_3)$ as the NEARIN heuristic.

For our running example, both heuristics differ in the selection of variant nodes to be integrated in the intermediate tour, but result in the same optimized path $\rho^{min}_{G_V}$ which is in our case equal to the optimal solution.

5.4 Implementation and Evaluation

In this section, we shortly present the prototypical implementation of our framework for model-based regression testing of variants and versions of variants. Furthermore, we describe the evaluation of our retest test selection as well as reuse-optimizing prioritization technique, where we use the three evolving delta-oriented product lines introduced in Chapt. 3 as subject systems.

5.4.1 Prototype

For our model-based regression testing framework, we provide a prototypical tool support which is also realized as Eclipse[1] plug-ins using EMF.[2] The following plug-ins are specified for our framework based on corresponding meta models:

- `de.imotep.testgen.testsuite` – Plug-in for test-suite management.
- `de.imotep.testgen.testgoal.coverage` – Plug-in for test-goal management.
- `de.imotep.regression.testartifacts` – Plug-in for variant-specific test-artifact management.
- `de.imotep.regression` – Plug-in for model-based regression testing of evolving SPLs.

Similar to the plug-ins for delta-oriented test modeling (cf. Sect. 3.4) as well as the delta-oriented change impact analyses (cf. Sect. 4.3), the plug-ins facilitating the application of our framework for consecutively testing SPL versions are part of the tool support of the research project IMoTEP.[3] The meta models and, therefore, the plug-ins facilitate the improvements and extensions of our testing framework in the future, e.g., by incorporating test-suite minimization techniques. In the following paragraphs, we describe the meta models of the plug-ins.

[1] https://www.eclipse.org/, last access: May 31st, 2019
[2] https://www.eclipse.org/modeling/emf/, last access: May 31st, 2019
[3] http://www.dfg-spp1593.de/imotep/, last access: May 31st, 2019

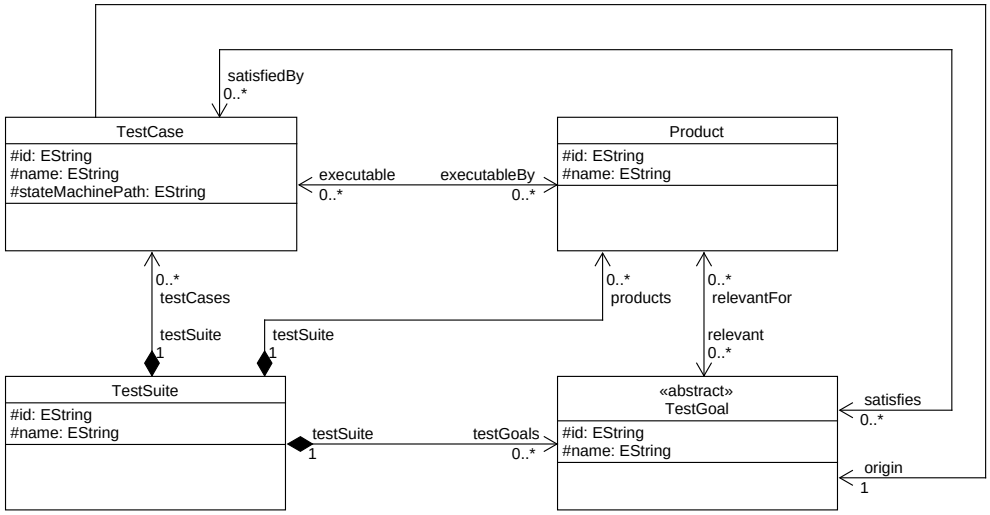


Figure 5.9: Meta Model of the Test-Suite Management Plug-In

Test-Suite Management. We require the plug-in `de.imotep.testgen.testsuite` for capturing test cases, test goals, and their relations in a test suite after their derivation by applying model-based testing techniques. All classes are used in the plug-in `de.imotep.regression.testartifacts` to allow for the specification of variant-specific test artifacts and in the plug-in `de.imotep.regression` to facilitate the definition of the shared test-artifact repository. The main class of the meta model shown in Fig. 5.9 is `TestSuite`. A `TestSuite` collects all created `TestCases`, all derived `TestGoals`, and a set of `Products` for which the `TestSuite` is valid. Each `TestCase` is related to the set of `TestGoals` it `satisfies` by traversing the respective state machine element via its `stateMachinePath`. A `TestCase` is further mapped to the `TestGoal` it was generated for by means of the relation `origin`. In addition, for each `TestCase` the set of `Products` is captured in order to specify on which `Product` a `TestCase` is executable via the relation `executableBy`. For each `TestGoal`, the set of `TestCases` traversing the `TestGoal` via the respective `stateMachinePath` is given by the relation `satisfiedBy`. Furthermore, the mapping of a `TestGoal` to the set of `Products` is specified via `relevantFor`. For each `Product`, we also capture the relations to `TestCases` and `TestGoals` to allow for the reasoning about the set of `TestGoals` which are `relevant` and about the set of `TestCases` which are `executable` for a given `Product`.

Test-Goal Management. The plug-in `de.imotep.testgen.testgoal.coverage` defines the types of test goals to allow for the incorporation of different standard test coverage criteria, e.g., all-state or all-transition coverage, and of our retest coverage criterion. The classes of the test goal types described in the following are applied in our framework for controlling the test-case generation and our retest test selection. The main class of the meta model depicted in Fig. 5.10 is `TestGoal`. We define six goal types which all inherit from the abstract class `TestGoal`, namely `StateGoal`, `TransitionGoal`, `StatePairGoal`, `TransitionPairGoal`, `TransitionStatePairGoal`, and `StateTransitionPairGoal`. Each subtype of `TestGoal` is mapped to the `State`, `Transition`, or a pair of both state machine elements the goal is derived for during the application of our framework. Please note, we use the four classes `StatePairGoal`, `TransitionPairGoal`, `TransitionStatePairGoal`,

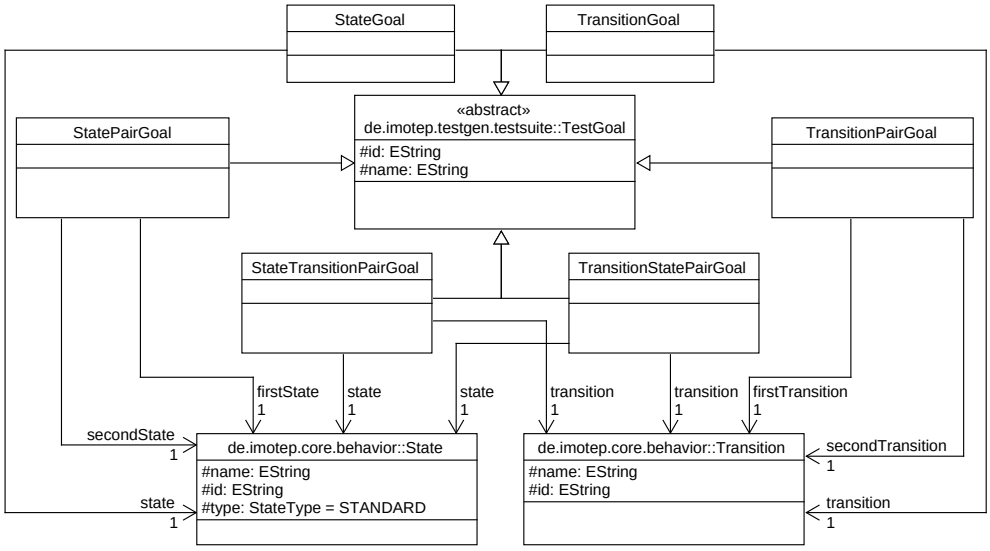


Figure 5.10: Meta Model of the Test-Goal Coverage Plug-In

and `StateTransitionPairGoal` for the derivation of retest test goals, where the name defines the potential combination and order of state machine elements to be traversed by a covering test case.

Variant-Specific Test Artifacts. We require the plug-in `de.imotep.regression.testartifacts` for capturing variant-specific test artifacts which are used for the application of our model-based regression testing framework. The main class of the meta model shown in Fig. 5.11 is `ProductTestArtifact`. A `ProductTestArtifact` captures a `StateMachine` as its `testModel`, a `TestSuite` to collect a set of `TestCases` as well as `TestGoals`, and a set of `SliceMappings`. For the description of the interrelation between the classes `TestSuite`, `TestCase`, and `TestGoal`, we refer to the already explained plug-in `de.imotep.testgen.testsuite`. A `SliceMapping` records for which `SlicingCriterion` a `Slice` is computed. For the completeness of the recording, a `SliceMapping` is also mapped to the `TestGoal` used as `SlicingCriterion` as well as the `Product`, the `ProductTestArtifact` is created for. To facilitate the incremental slice computation, a `ProductTestArtifact` has a relation to the variant-specific `DependencyGraph`. Furthermore, a `ProductTestArtifact` refers to the set of `newTestCases`, `reusableTestCases`, `retestTestCases`, and `retestTestGoals` via respective relations after the test process for the current variant under test has finished.

Model-Based SPL Regression Testing. The plug-in `de.imotep.regression` realizes our framework for model-based regression testing of evolving SPLs. The main class of the meta model depicted in Fig. 5.12 is `RegressionTestManager`. A `RegressionTestManager` captures the `HigherOrderDeltaRepository` documenting the evolution history, the `SlicingManager` for change impact analysis, and the sets of version-specific `FeatureConfigurationManager`, `StateMachineDeltaRepository`, and `DeltaDependencyGraph`. In addition, a `RegressionTestManager` contains the shared `TestArtifactRepository` which, in turn, records all created `TestCases` and derived `TestGoals` via a respective `TestSuite` as well as the set of `SliceMappings`. Both, the `TestSuite` and the set of `SliceMappings` are updated after each testing process of a variant or version of a variant. To initialize the workflow of our model-based regression testing framework, the methods `initRegressionTest-`

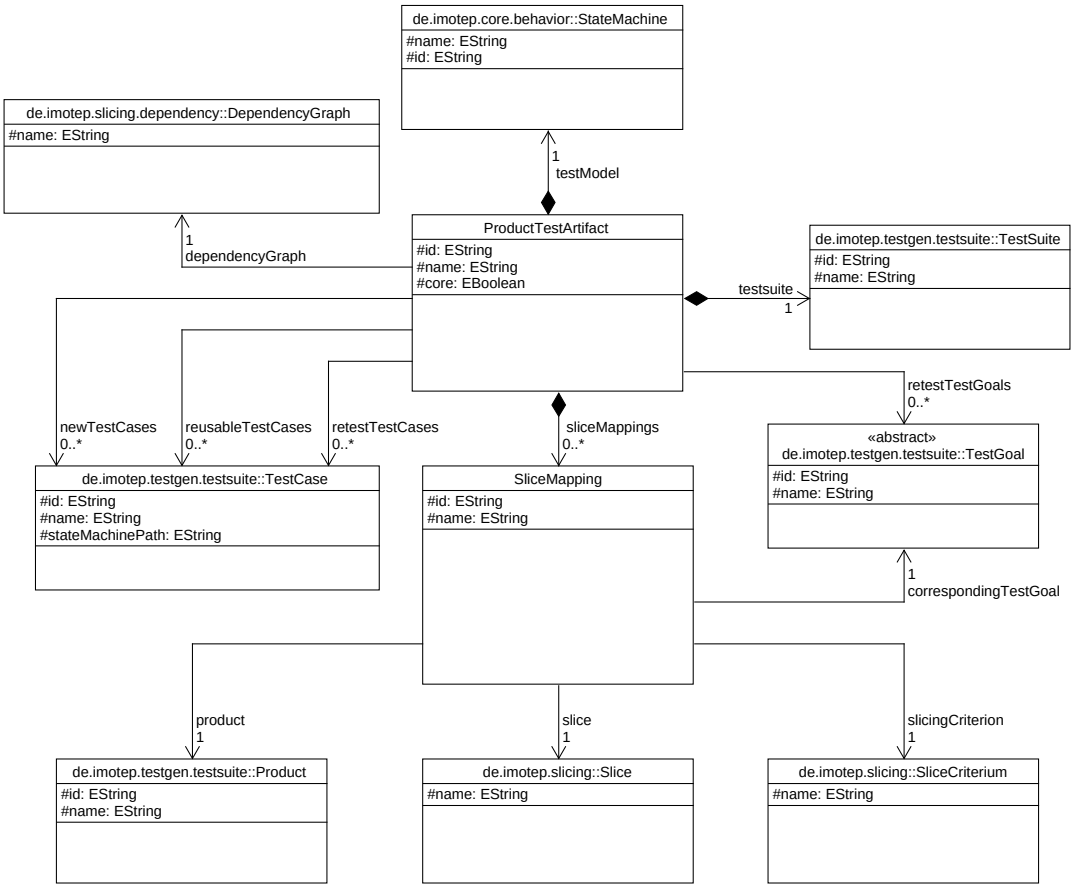


Figure 5.11: Meta Model of the Regression Testing Test Artifacts Plug-In

`ManagerMultipleVersions()` and `initializeProductOrderToBeTested()` have to be executed. The first method controls the loading of all version-specific `FeatureConfigurationManager` and `StateMachineDeltaRepository` as well as of the `HigherOrderDeltaRepository` on which the execution of our framework is based on. The second method sets the testing order for the testing process of the initial SPL version (cf. Sect. 5.3). After the initialization, we start the automated workflow of our regression testing framework by executing the method `startRegressionTesting()`.

Each version of an SPL that is tested by the prototypical implementation of our framework is represented by the class `SPLVersionUnderTest` and captured by the `RegressionTestManager`. A `SPLVersionUnderTest` comprises the `FeatureModel` specifying the variability and commonality of the SPL version and also the set of `ProductConfigurations` representing the variants under test. Furthermore, each `SPLVersionUnderTest` is mapped to the version-specific instances of `FeatureConfigurationManager`, `DeltaDependencyGraph`, and `StateMachineDeltaRepository` via respective relations. During the regression testing of an `SPLVersionUnderTest`, the relation `toBeTestedProducts` refer to set of not yet tested `ProductConfigurations`, whereas the relation `testedProducts` collects the set of `ProductConfigurations` which are already tested by exploiting the test artifacts and test results of preceding testing processes. When stepping to the next `SPLVersionUnderTest`, the categorization determined based on the higher-order delta application reasoning is cap-

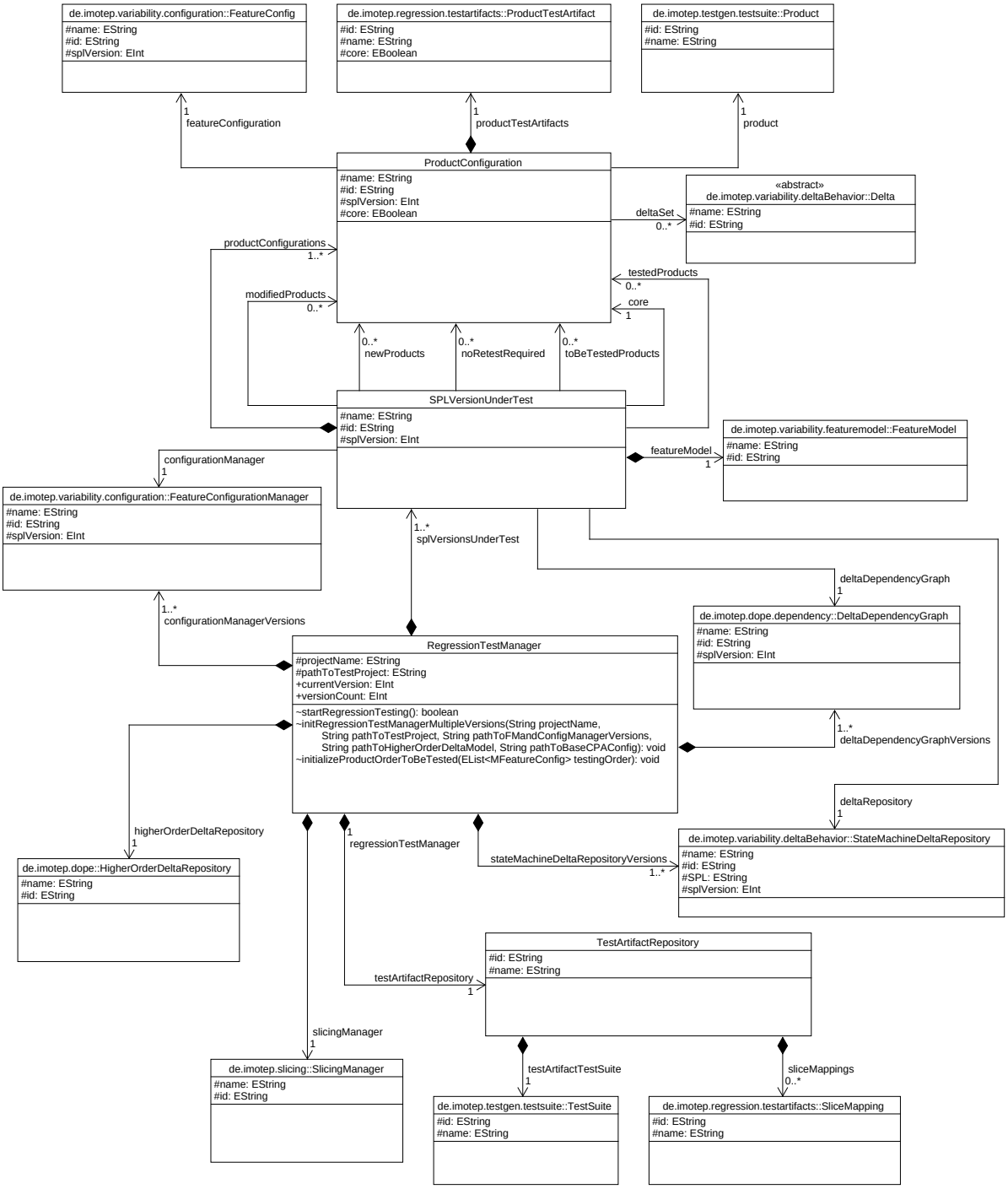


Figure 5.12: Meta Model of the Regression Testing Plug-In

tured by the relations `newProducts`, `modifiedProducts`, and `noRetestRequired`. The categorization is exploited during the execution of our framework for regression testing of variants and versions of variants. Each variant to be tested is represented by the class `ProductConfiguration`. A `ProductConfiguration` is mapped to its `FeatureConfig`, its `Product`, and its set of `Deltas` for a unique identification. In addition, a `ProductConfiguration` captures its set of variant-specific test artifacts via `ProductTestArtifact` used during the automated testing workflow of our framework.

Based on all realized plug-ins for delta-oriented test modeling as well as change impact analysis, and those plug-ins described in this section, we are able to perform model-based regression testing for evolving delta-oriented SPLs. The complete prototype of the regression testing framework is provided online.[4]

5.4.2 Evaluation of the Model-Based Regression Testing Framework

In this section, we present the evaluation of our framework to validate its efficiency and effectiveness. First, we formulate the research questions and describe the methodology of the evaluation. Second, we present and discuss our obtained results and the threats to the validity of our evaluation.

Research Questions and Methodology

The evaluation of our model-based regression testing framework is defined as *controlled experiment*, where we apply its prototypical implementation to the three evolving subject SPLs (cf. Sect. 3.4.2). Please note that we only evaluate our framework w.r.t. the retest test selection and abstract from the potential evaluation of the process of test modeling and test-case generation. For the documentation of the research methodology, we followed the guidelines defined by Wohlin et al. [WHH03; WRH+12] as well as Juristo and Moreno [JM13]. To conduct the experiment, we formulate the following research questions (RQ) to be answered.

RQ1 What is the *influence* of varying testing orders on retest decisions during model-based SPL regression testing?

RQ2 Do we achieve a *reduction* of test cases to be executed from our retest test selection compared to retest-all [YH12]?

RQ3 Do we ensure *effectiveness* with our retest test selection compared to retest-all [YH12]?

To answer the defined research questions, we again determine both *qualitative* and *quantitative* data. The higher-order delta test models of the three subject systems and also the analysis artifacts, i.e., slices, (retest) test goals, (retest) test cases etc., created during the experiment define the qualitative data. Just as for the controlled experiment of the delta-oriented change impact analyses (cf. Sect. 4.3), we executed the prototypical implementation of our framework on a machine with 32 Intel Xenon E5 (3.1GHz) cores and 50GB RAM running Ubuntu 18.04.1 LTS x86_64 as operating system. By applying metrics, e.g., number of test cases, to the computed testing artifacts, we obtain quantitative data. We exploit the quantitative data to facilitate a *hypothesis confirmation*, where we use the defined research questions as hypothesis. To investigate whether a hypothesis can be confirmed, we apply the following data analysis and research methodology.

For the investigation how distinct testing orders may influence our framework, we aim to answer the research question **RQ1** to provide a preliminary reasoning whether there is an influence of testing orders w.r.t. the exploitable test artifact reuse potential or not. To answer **RQ1**, we examine

[4] `https://github.com/SLity/mbtSPLregression`

the total number of changes between subsequently tested variants and further the number of retest test goals derived by our change impact analysis. We incorporate the total number of changes as our slicing-based impact analysis takes changes into account for detecting slice differences which, in turn, are used to derive retest test goals. We focus on the number of retest test goals and not on the number of selected test cases as retest test goals indicate the retest potential to reason about. In addition, the retest test selection depends on the set of generated test cases such that distinct sets of test cases will influence the outcome of the controlled experiment. The evaluation based on **RQ1** denotes solely a preliminary evaluation. As our framework is applicable independently from a certain prioritization technique providing an optimized testing order, a comprehensive evaluation of distinct testing orders and their influence is out of scope of this thesis. This includes also the evaluation of the effectiveness of varying testing orders applied for SPL regression testing. The analysis and comparison of prioritization techniques regarding their efficiency and effectiveness is an open research topic to be answered in future research.

We apply different testing orders for testing the initial SPL versions of the three evolving subject SPLs which are computed based on (1) our prioritization technique for computing reuse-optimized testing orders (cf. Sect. 5.3), (2) existing prioritization techniques [ALL+17; ATM+14; HPP+14] focusing on the dissimilarity between variants under test, and (3) random generation. We select those existing techniques [ALL+17; ATM+14; HPP+14] to compare our results against, as they are applicable due to the available artifacts used in this experiment as required input, i.e., delta sets and feature configurations. Thus, we apply the feature similarity [ATM+14] and delta similarity technique [ALL+17] of Al-Hajjaji et al. as well as the global maximum distance (GMD) and local maximum distance (LMD) approach from Henard et al. [HPP+14]. We refer to the discussion about related work in Sect. 5.5 for a description of those techniques. For the random testing orders which are used as a *baseline*, we generate 100 orders for each of the three subject SPLs. Some of the existing techniques, e.g., the LMD strategy of Henard et al. [HPP+14], are influenced by the initial, yet unordered order of variants to compute their prioritization. To cope with this influence, we apply for each prioritization technique the random generated testing orders as input. For an evaluation of our prioritization technique w.r.t. the quality of achieved approximations as well as the performance to find good approximations of reuse-optimized testing orders, we refer to Lity et al. [LAT+17].

For the evaluation of our model-based regression testing framework w.r.t. the retest test selection, we have to answer the research questions **RQ2** and **RQ3**. The question **RQ2** investigates the efficiency of our retest test selection by means of a reduced set of executed test cases. The question **RQ3** examines the effectiveness of the selection technique in terms of the fault detection rate. We answer **RQ2** by comparing the number of retest test cases selected and generated by our framework against the number of test cases reexecuted based on the retest-all strategy [YH12] used as a *baseline* to check for a reduction of test-case executions. Following the retest-all strategy [YH12], all test cases that are categorized as reusable when stepping to the next variant under test are reexecuted for a retest. This holds also for variants that are categorized as unchanged based on our variant set change impact analysis when stepping to the next SPL version under test such that we reexecute all test cases of the previous version of the variant. The set of reusable test cases reexecuted by the retest-all strategy includes those test cases from previously tested variants which were generated for standard transition test goals as well as test cases which were generated for ensuring retest test goal coverage. The set of reusable test cases which are selected by our framework is a subset of the

retest-all test suite, but the retest test suite determined by our framework further comprises newly generated retest test cases which are not taken into account for retest-all. We generate additional retest test cases to ensure 100% retest coverage which cannot be achieved by the retest-all strategy in general. However, the comparison of the retest-all strategy used as baseline and our framework w.r.t. the number of reexecuted test cases is reasonable as we are still able to compare both retest test suites to identify which one reexecutes less test cases. Therefore, we are still able to reason about whether a reduction of the overall test effort is achievable based on the application of our framework compared to retest-all.

We apply our framework on the complete variant set for each SPL and its versions and do not apply sampling strategies [VAT+18] in our controlled experiment. The abstraction from sampling strategies [VAT+18] ensures that our results are not influenced and, therefore, dependent on a computed subset of variants to be tested. Each subject SPL is tested by consecutively testing its SPL versions following the workflow of our model-based regression testing framework (cf. Sect. 5.1). We start with the respective initial versions, where we apply the NEARIN heuristic (cf. Sect. 5.3) to define a reuse-optimized testing order. The NEARIN heuristic provides the best approximations for reuse-optimal testing orders which can be seen in the results of the investigation of **RQ1**. We do not apply the optimal testing orders as with an increasing number of variants to be tested, the computation of an optimal solution is not practical. For the remaining SPL versions of a subject SPL, we exploit the testing order obtained based on the incremental delta set derivation for regression testing of new variants under test. As coverage criterion to guide standard test-case generation, we apply all-transition coverage [UL06]. Hence, we also use transition test goals as slicing criteria for our slicing-based change impact analysis. For the test-case derivation, we implemented a prototypical generator that performs event simulation as well as incremental depth-first search to generate covering test cases for given standard as well as retest test goals during the regression testing of variants and versions of variants.

To answer **RQ3**, we evaluate the fault detection rate of the retest set of test cases determined by our retest test selection compared to the results of the retest-all strategy [YH12]. Unfortunately, the three delta-oriented evolving SPLs do not have a real fault history to be used for the evaluation. To tackle this drawback, we perform a fault simulation to facilitate a reasoning about the effectiveness of our framework. Due to absence of real fault information for the three systems, fault simulation can either be achieved based on the application of model-based mutation testing for state machines [ABJ+15; LS14] or by generating random artificial faults. Furthermore, for the simulation, the types of faults have to be specified as the type has an influence on the result and soundness of the evaluation. As our framework focuses on the retest of already tested behavior which is potentially erroneously influenced by changes between subsequently tested variants and versions of variants, we use erroneous execution interactions between artifacts as fault type to be simulated. Hence, a simulated fault represents an erroneous execution interaction, e.g., between two transitions, which is caused by changes and their impact on common behavior. An example for such a fault type is the erroneous assembling of development artifacts on the implementation level such that unintended, yet flawed feature interactions are introduced.

By focusing on this type of faults to be simulated, the application of existing mutation testing techniques for state machines [ABJ+15; LS14] is prevented. Those techniques facilitate solely syntactic mutations simulating a distinct fault type, where, for instance, behavior represented by a

transition is completely missing based on its removal or is newly specified based on the reconnection of a transition to a different source or target state. Furthermore, our framework is defined for a black-box test setting as described in Sect. 2.1, where we have no access to the source code. Hence, we are not able to incorporate source code changes, which are solely made on the implementation level and have no respective changes in the variant-specific test models, for the change impact analysis to detect their impact and to find potential faults w.r.t. such changes. As we cannot determine the impact of potentially erroneous source code changes, we also abstract from this type of faults in our evaluation.

Therefore, we generate random artificial faults representing erroneous execution interactions, where we incorporate the differences between subsequently tested variants by means of change operations captured in respective regression deltas. We derive faults by combining elements which are added or removed by change operations, i.e., added transitions as well as source and target states of removed transitions, with randomly chosen transitions from the current state machine test model under consideration. Thus, a fault is defined as pair of elements $(elem_{op_\delta}, elem')$ and denotes that the interaction between both elements during the system execution is not correctly implemented such that the first element $elem_{op_\delta}$ given by the change operation erroneously influences the execution of the second randomly chosen element $elem'$. For the detection of such simulated faults, a test case tc to be retested has to traverse both elements in the correct order via its state machine path ρ^{tc}_{sm}. The reasoning about fault detection via the coverage is sufficient as those faults are artificial and have no real mapping to the implementation. For each variant as well as version of a variant, we generate a respective set of simulated faults. Depending on the size of such fault sets, we derive a maximum of distinct data sets, where each time a random selection of 10% of the faults is made to obtain random fault data sets. We execute the retest set of test cases of our framework and also of the retest-all strategy [YH12] on each random data set to assess the fault detection rate.

Results

We present and discuss the results of our evaluation individually for the defined research questions.

RQ1. In Tab. 5.2, we summarize the results of the influence of different testing orders regarding the total number of changes between subsequently tested variants. In addition, we summarize the results of the test-order influence on our framework by means of the number of derived retest test goals in Tab. 5.3. For both tables, each column represents one of the prioritization techniques we applied in the controlled experiment. We provide average values for the total number of changes as well as for the total number of retest test goals w.r.t. the set of variants under test of the initial versions of the three subject SPLs and the respective 100 testing orders of each prioritization technique determined for the execution of the experiment.

Based on the results for the total number of changes, we can see that the varying testing orders have a respective impact. The dissimilarity-based prioritization techniques [ATM+14; ALL+17; HPP+14] compute testing orders where the number of changes is at least three times larger than the numbers determined for the reuse-optimal solutions in the second column. The random testing orders provide better results regarding the total number of changes compared to the existing techniques, but, similar to those techniques, the numbers are still two to three times larger than the reuse-optimal solutions. In contrast, our techniques compute testing orders such that the number of changes solely slightly differ compared to the optimal solutions. For the three subject SPLs and

Table 5.2: Results of the Impact of Testing Orders Regarding the Total Number of Changes for **W**iper, **V**ending **M**achine, and **M**ine **P**ump

SPL	Optimal	Greedy	Look-Up Greedy	NEARIN	FARIN	GMD [HPP+14]	LMD [HPP+14]	Feature [ATM+14]	Delta [ALL+17]	Random
W_{θ_0}	25	25.46	25.46	25.46	25.96	78.99	77.93	70.07	75.65	58.66
VM_{θ_0}	62	63.68	62.34	63.30	67.06	282.77	278.66	218.14	218.38	208.44
MP_{θ_0}	81	82.74	82.74	82.26	89.26	429.89	444.01	354.03	364.71	310.50

Table 5.3: Results of the Impact of Testing Orders Regarding the Number of Retest Test Goals for **W**iper, **V**ending **M**achine, and **M**ine **P**ump

SPL	Optimal	Greedy	Look-Up Greedy	NEARIN	FARIN	GMD [HPP+14]	LMD [HPP+14]	Feature [ATM+14]	Delta [ALL+17]	Random
W_{θ_0}	24.14	24.31	24.31	24.50	24.65	66.36	65.57	61.85	64.50	51.52
VM_{θ_0}	41.51	43.05	42.53	42.38	45.13	119.36	113.93	108.46	107.22	102.73
MP_{θ_0}	33.86	34.86	34.86	34.84	37.73	77.19	80.03	74.21	77.19	66.84

their initial SPL versions, the NEARIN heuristic provides the best approximation of the optimal solution in the most cases.

As the retest test goal derivation is dependent on the difference between subsequently tested variants by means of change operations captured in respective regression deltas, we can see a similar impact of varying testing orders on the number of derived retest test goals. The coverage-driven dissimilarity-based testing orders as well as the random testing orders result in numbers that are two times larger as for the optimal testing order which provides the smallest number of derived retest test goals. Again, our strategies provide a good approximation compared to the optimal testing order such that the respective testing orders facilitate the derivation of the smallest numbers of retest test goals, where the NEARIN heuristic provides the best approximation in most cases.

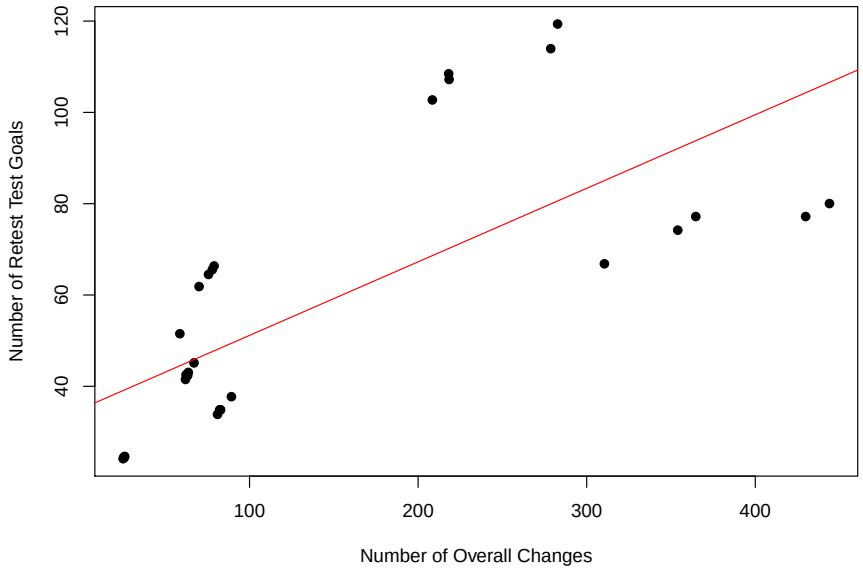


Figure 5.13: Relation of the Number of Overall Changes to the Number of Retest Test Goals

In summary, the investigation of **RQ1** represents a preliminary evaluation of the impact of varying testing orders on SPL regression testing in terms of derived retest test goals indicating retest potentials to reason about. By taking all results into account, we derive the tendency that the smaller the number of changes, the smaller is the number of derived retest test goals. The tendency is also shown by the regression line in Fig. 5.13, where we related the number of overall changes to the number of retest test goals in a scatter plot. To substantiate this tendency, we have to perform more controlled experiments with additional SPLs in the future. Furthermore, our model-based regression testing framework performs better in terms of less retest test goals to be covered if the changes between variants are reduced to a minimum, e.g., based on the application of the NEARIN heuristic. However, this scenario may prevent from an early fault detection rate that gets increased by following a dissimilarity-based strategy [ATM+14; ALL+17; HPP+14]. Hence, there is a trade-off between both objectives to cope with, e.g., by applying multi-objective optimization techniques, representing an open question which is out of scope of this thesis, but postponed to future research. Please note, as the NEARIN heuristic computes the best approximation for the reuse-optimized testing orders, we will apply the heuristic to provide testing orders of the initial SPL versions of the three subject SPLs in our controlled experiment of research question **RQ2**.

Table 5.4: Results of Retest Test Selection for **W**iper, **V**ending **M**achine, and **M**ine **P**ump (⌀ = Average, N = New, T = Transition, Re = Retest, R = Reuse, O = Obsolete, S = Select)

SPL	⌀ Test Goals (Transition + Retest)	⌀ Test Cases (New + Reuse + Obsolete)	⌀ Retest (Selected + New)	⌀ Retest All	⌀ Retest Coverage (%)	⌀ TS_v^{Re} / TS_v^{R} (%)
W_{θ_0}	35.5 (17.5+18.0)	44.1 (8.0+16.0+20.1)	14.6 (9.5+5.1)	16.0	54.4	9.4
W_{θ_1}	104.8 (22.5+82.3)	182.4 (23.1+46.1+113.1)	42.6 (21.3+21.4)	46.1	49.4	8.2
W_{θ_2}	72.8 (26.3+46.4)	285.8 (12.3+76.4+197.1)	34.7 (23.4+11.3)	76.4	41.8	120.4
W_{θ_3}	155.5 (30.3+125.1)	523.9 (14.8+117.2+391.9)	71.4 (57.1+14.3)	117.2	77.4	64.4
W_{θ_4}	81.5 (32.3+49.1)	780.4 (3.0+114.2+663.2)	72.0 (69.3+2.7)	114.2	85.6	84.3
VM_{θ_0}	55.9 (14.7+41.2)	68.3 (3.3+12.3+52.8)	8.1 (6.3+1.8)	12.3	77.6	52.2
VM_{θ_1}	94.7 (17.9+76.8)	142.8 (5.0+27.1+110.6)	15.9 (11.7+4.3)	27.1	66.9	70.4
VM_{θ_2}	124.1 (18.7+105.3)	395.2 (2.9+31.4+360.9)	22.8 (19.9+2.8)	31.4	92.0	37.8
VM_{θ_3}	16.5 (16.5+0.0)	508.0 (0.0+25.5+482.5)	0.0 (0.0+0.0)	25.5	–	–
VM_{θ_4}	91.1 (21.1+70.0)	535.64 (3.4+27.4+504.8)	13.6 (10.5+3.1)	27.4	73.8	101.2
VM_{θ_5}	96.7 (22.8+73.9)	670.3 (2.2+34.9+633.1)	14.7 (12.7+2.0)	34.9	86.4	137.3
VM_{θ_6}	278.0 (31.9+246.2)	1140.8 (14.1+53.9+1072.8)	53.0 (39.9+13.1)	53.9	72.1	1.7
MP_{θ_0}	65.3 (33.5+31.8)	51.6 (5.1+18.5+28.0)	9.6 (7.6+1.9)	18.5	64.9	93.5
MP_{θ_1}	64.5 (37.5+27.0)	103.6 (3.8+31.4+68.4)	6.8 (3.8+3.0)	31.4	49.3	361.5
MP_{θ_2}	58.5 (40.3+18.1)	190.1 (3.0+27.5+159.6)	6.5 (5.8+0.8)	27.5	74.0	317.1

RQ2. In Tab 5.4, we summarize the results of our framework regarding the retest test selection based on its application for the three evolving subject SPLs (cf. Sect. 3.4). Please note, all values in Tab. 5.4 are given as average values (⌀) w.r.t. the size of the version-specific variant sets. In the second column, we provide the total number of derived test goals per variant under test divided into the number of standard transition test goals as well as retest test goals. The third column captures the number of test cases per variant composed by the number of new, reusable, and obsolete test cases. The test cases categorized as new are further aggregated based on the set of new test cases generated to cover standard test goals as well as newly generated retest test cases. In the fourth column, we provide the number of retest test cases selected and newly generated based on the application of our retest test selection. The fifth column comprises the number of test cases reexecuted by following the retest-all strategy. In the sixth column, we provide the percentage of retest coverage ensured based on the retest-all strategy. The last column captures the relation between the number of test cases reexecuted via retest-all and the number of test cases retested via our framework to reason about the percentage reduction of executed test cases.

We answer **RQ2** by comparing the retest test suite determined by our framework and the retest test suite derived following the retest-all strategy. As we can see, our framework retests less test cases than retest-all which holds for all three subject SPLs and their respective versions. For the versions θ_0 and θ_1 of the Wiper SPL as well as the version θ_6 of the Vending Machine SPL, we obtain the lowest reduction of executed test cases, where retest-all solely selects around 9% more test cases for the Wiper and around 2% more test cases for the Vending Machine SPL compared to our framework. We investigated those three cases and their respective evolution steps. In each case, the evolution step introduced functionality that strongly interacts with the main functionality of the systems

and, therefore, has a large influence on already tested functionality between versions of variants. For instance, the evolution step of version θ_6 of the Vending Machine SPL introduced the choice of milk for other beverages than cappuccino [NLS18]. As the choice of milk has an influence on the main behavior, i.e.,the offering of different beverages, an increased number of retest test goals are derived as we can see in Tab. 5.4. Hence, to ensure retest test coverage, a larger number of test cases was selected such that we achieved a low reduction of test-case executions compared to retest-all.

In contrast to the low reduction, for the SPL versions θ_5 of the Vending Machine SPL as well as θ_2 of the Wiper SPL, the retest-all strategy reexecutes around 120% more test cases than our framework. In addition, for the SPL versions θ_1 and θ_2 of the Mine Pump SPL, our framework achieves the largest reduction, where for retest-all around 300% more test cases are executed. Based on the large difference between the results of the Mine Pump SPL versions and the other results, we, again, examined their respective evolution steps. As mentioned in the description of the three subject SPLs in Sect. 3.4.2, the subject systems may comprise variables in order to provide also a synchronization between concurrent regions via shared variables. This fact influences our framework in both versions θ_1 and θ_2 of the Mine Pump SPL. In both versions, new functionality is introduced and captured in separate regions in the state machine test models. However, the synchronization between the new behavior and the existing behavior which was already tested for respective variant versions is defined via shared variables. Our slicing-based impact analysis cannot identify respective data-specific changes in the execution dependencies as we focus solely on control dependencies which results in missing retest test goals and a smaller number of selected test cases to be reexecuted.

On average, the retest-all strategy retests circa 90% more test cases. We mainly achieve this reduction by exploiting the results of the variant set change impact analysis, where we identify unchanged variants for which no retest has to be applied. For those variants, our framework identifies no retest potentials as nothing has changed, but for retest-all, all reusable test cases have to be reexecuted. A special case is given in version θ_3 of the Vending Machine SPL, where due to the respective evolution step solely variants are removed and all remaining variants are categorized as unchanged such that we do not have to perform a retest at all.

By abstracting from the unchanged variant versions for the comparison, i.e., we take solely the modified and new versions of variants into account, our framework reexecutes a similar amount of test cases as retest-all for retesting a variant version. There are also some cases, where our framework selects slightly more. However, our retest test selection is guided based on the retest coverage criterion. The retest test suites determined by our framework ensure 100% coverage for the regression testing of variants and versions of variants. As we can see in Tab. 5.4, the retest-all strategy does not achieve the complete coverage for any subject SPL and its versions. We deduce that the effort for guaranteeing retest coverage for the retest of modified and new versions of variants is acceptable, especially, as we do not retest unchanged variants. In addition, for the version θ_1 of the Wiper SPL, where all variants are modified based on the addition of a core delta, our framework still reexecutes less test cases than the retest-all strategy.

Furthermore, by focusing on the composition of the retest test suite determined by our framework in terms of selected reusable and newly generated retest test cases, we see that more reusable test cases are selected as newly generated to achieve retest coverage. With the exception of the versions θ_1, θ_2, and θ_3 of the Wiper SPL as well as version θ_6 of the Vending Machine SPL, we solely generate a small amount of retest test cases. For all other SPL versions of the subject SPLs, we gener-

ate two to five new retest test cases on average. Those retest test cases represent specific execution scenarios for testing the artifact interactions defined by retest test goals. The test-case generation requires further effort at this point, but test cases are generated once and are potentially reusable and selectable for a retest in subsequent testing processes of variants and versions of variants.

To summarize, the results of the application of our model-based regression testing framework w.r.t. the retest test selection to the three evolving subject SPLs show its efficiency by means of an achieved reduction of the test effort by reducing the number of executed test cases (**RQ2**). By exploiting the delta-oriented test-modeling formalism as well as the delta-oriented change impact analyses, the retest test selection facilitates the derivation of smaller retest test suites compared to the retest-all strategy. In contrast to retest-all, our framework further ensures retest goal coverage. However, the set of test cases to be retested may contain some redundancy in terms of standard as well as retest test goal coverage. Hence, there exists potential for further test-effort reduction based on the application of test-suite minimization or optimization techniques [YH12] which is out of scope of this thesis, but postponed to future research. In addition, our controlled experiment has shown that the incorporation of data dependencies is important for a comprehensive change impact analysis and guided retest test selection. To improve our results and, therefore, to improve our testing framework, the incorporation of data dependencies in our slicing-based change impact analysis is a reasonable step for future work.

RQ3. In Tab. 5.5, we summarize the results for the evaluation of the effectiveness in terms of the fault detection rate. The second column captures the average number of simulated faults per variant of the respective SPL version under test. In the third column, we provide the number of fault sets which are derivable for all variants to be tested of an SPL version by randomly selecting 10% of the simulated faults. The fourth column comprises the size of the randomly selected fault sets. In the fifth column, we present the average ratio of undetected (alive) and detected (dead) faults based on the retest test suites determined by our framework, whereas the last column presents the average ratio of alive and dead faults based on the results of the retest-all strategy. Please note, for the controlled experiment regarding the fault detection rate, we do not take unchanged versions of variants into account. Hence, for version θ_3 of the Vending Machine SPL, no data regarding the fault detection rate is acquired.

Based on the presented results, we can see that our retest test selection achieves a good fault detection rate for the three evolving subject SPLs. Compared to the retest-all strategy, our technique performs even better as we achieve a higher ratio of dead to alive faults. However, we must relativize this result as (1) we have to incorporate the results for the efficiency investigation shown in Tab. 5.4 and (2) we focus on a specific fault type in our evaluation for **RQ3**, namely erroneous artifact interactions. As discussed for research question **RQ2**, our framework retests a similar amount of test cases on average for modified and new versions of variants to be tested compared to retest-all when we abstract from unchanged variants. Although the numbers of reexecuted test cases are similar, the respective retest test suites are differently composed as described in Sect. 5.2 as well as in the research methodology of the framework evaluation. The retest test suites obtained for retest-all represent the sets of reusable test cases including those test cases from previously tested variants which were generated for standard transition test goals as well as test cases which were generated for ensuring retest test goal coverage. In contrast, the retest test suites determined by our framework denote subsets of the reusable test cases and further newly generated retest test cases which are required

Table 5.5: Results of the Framework Effectiveness for **W**iper, **V**ending **M**achine, and **M**ine **P**ump Compared to Retest-All (⌀ = Average, # = Number)

SPL	⌀ Faults	# Fault Sets	Size Fault Sets	Framework ⌀ Alive/⌀ Dead	Retest-All ⌀ Alive/⌀ Dead
W_{θ_0}	37.4	22	3	0.4/2.6	1.5/1.5
W_{θ_1}	112.5	26	3	0.1/2.9	1.6/1.4
W_{θ_2}	265.0	100	18	5.5/12.5	9.8/8.2
W_{θ_3}	148.2	57	6	0.5/5.5	0.7/5.3
W_{θ_4}	366.3	100	21	8.0/13.0	9.5/11.5
VM_{θ_0}	41.0	30	4	0.2/3.8	0.9/3.1
VM_{θ_1}	281.0	100	12	1.3/10.7	2.7/9.3
VM_{θ_2}	177.1	68	8	0.4/7.6	0.9/7.1
VM_{θ_3}	–	–	–	–/–	–/–
VM_{θ_4}	98.3	68	7	3.4/3.6	4.2/2.8
VM_{θ_5}	78.3	38	4	3.7/0.3	3.3/0.7
VM_{θ_6}	242.4	75	13	0.7/12.4	2.5/10.5
MP_{θ_0}	81.0	22	3	0.8/2.2	1.5/1.5
MP_{θ_1}	262.5	100	22	9.3/12.7	15.1/6.9
MP_{θ_2}	319.3	33	4	3.5/0.5	3.0/1.0

to ensure retest coverage for the current variant under test. As we achieve a better fault detection rate independent from whether we retest less or similar test cases than retest-all, we assume that the small number of newly generated retest test cases increases the fault detection rate of erroneous artifact interactions. This fact is supported by the fault type we focus on for the evaluation as newly generated retest test cases denote representative executions of variants to be retested validating that no unexpected behavior is implemented based on new or changed dependencies/interactions. Therefore, we infer that our framework allows for a good detection rate of faults related to erroneous artifact interactions as we guarantee retest test coverage by generating additional retest test cases.

Furthermore, we can see some special cases in the results shown in Tab. 5.5. For version θ_4 of the Vending Machine SPL, the ratio of dead to alive faults is rather balanced, but our framework still detects more faults than retest-all on average. In contrast, for version θ_5 of the Vending Machine SPL, the results of our framework and of the retest-all strategy show that only a small number of the simulated faults are detected, where retest-all provides a slightly better detection rate. We can see similar results for version θ_2 of the Mine Pump SPL, where, again, retest-all performs slightly better. By examining the undetected simulated faults, we found out that those faults representing erroneous artifact interactions were seeded in model parts, where changes between state machine test models of subsequently tested variant versions had no impact such that our slicing-based change impact analysis could not identify them to guide the retest test selection. In addition, for the three SPL versions, we achieved the largest reduction in the number of test cases to be retested compared to retest-all for all evolving subject SPLs which has also an influence on the detection rate.

In summary, the retest test selection integrated in our model-based regression testing framework for testing variants as well as versions of variants achieves a good fault detection rate w.r.t. the simulated erroneous artifact interactions. Compared to retest-all, the results show that our framework

is more effective w.r.t. the simulated fault type even with a reduced set of test cases to be reexecuted based on the fulfillment of the retest test coverage. Furthermore, we assume that erroneous artifact interactions are occurring between execution-related artifacts, e.g., transitions that are subsequently executed to perform a certain task, such that our slicing-based change impact analysis will identify changed execution dependencies indicating retest potentials that are retested by our framework.

5.4.3 Threats to Validity

For the evaluation of our model-based SPL regression testing framework as well as of our prioritization technique, the following threats to validity arise.

The selection of the three subject systems is a potential threat for the controlled experiment of our model-based SPL regression testing framework. The selection of an evaluation subject is a general drawback when applying a controlled experiment as it influences the potential to generalize the results for other systems. Similar to the controlled experiment for the delta-oriented change impact analyses (cf. Sect. 4.3), the three evolving SPLs provide different evolution and modeling characteristics influencing the results of our framework when applied to those systems. Therefore, based on the obtained results, we assume that they are, up to a certain extent, generalizable also to other evolving delta-oriented SPLs. However, we must substantiate this assumption by performing more experiments.

Furthermore, the size of of the systems by means of number of variants may be another potential threat. Compared to real-world SPLs, the three subject systems are rather small. However, the size of real-world SPLs is a general challenge in SPLE to be managed by SPL development as well as testing techniques [PBvdL05; McG10]. To cope with this challenge, sampling strategies are applied to determine a representative subset of variants [VAT+18], e.g., to perform testing [JHF12; AKT+16a]. Our framework is also applicable to a sample of variants such that it also facilitates efficient model-based regression testing of larger evolving SPLs based on a given representative subset of variants which has to be substantiated by performing more experiments with larger evolving SPLs.

For the preliminary evaluation of the impact of varying testing orders on SPL regression testing in terms of derived retest test goals representing retest potentials to reason about, the computation of the testing orders for the existing dissimilarity-based techniques [ALL+17; ATM+14; HPP+14] is a potential threat. As we did not have the access to all of the original prototypes, we reimplemented the respective technique if necessary. However, for the reimplementation, we have strictly adhered to the algorithms defined in the corresponding papers.

Furthermore, the preliminary assessment shows that varying testing orders influence the results of our framework such that the selection of the NEARIN heuristic to compute the testing orders for regression testing of the initial SPL versions is a potential threat. The NEARIN heuristic provided the best approximations regarding reuse-optimized testing orders, whereas the early fault detection rate is rather decreased compared to the dissimilarity-based prioritization techniques [ALL+17; ATM+14; HPP+14]. However, to find an optimal trade-off between those contrary objectives is out of scope of this thesis and left open for future research.

Our model-based regression testing framework is, in general, applicable independently from a certain testing order. Nevertheless, the computation of an optimized testing order requires time which we neglect for the evaluation of our framework representing a potential threat. However, as described in the research methodology, we assume a testing order to be given and omit the

comparison of prioritization techniques as it is out of scope of this thesis. Therefore, we refer to the respective papers for an analysis of the required runtime for computing optimized testing orders [ALL+17; ATM+14; HPP+14; LAT+17].

The choice of the coverage criterion used to guide testing steps such as the test-case generation is a potential threat. This is a general drawback of model-based testing [UL06] and is not specific for our regression testing framework. As commonly used in model-based testing [UL06; UPL12], we applied all-transition coverage as coverage criterion. However, other criteria or even more complex ones could also be incorporated in our framework.

In this context, the applied test-case generator is also a potential threat. Depending on the applied generator or derivation technique, the set of test cases may differ. The set of generated test cases builds the basis for our retest test selection such that different test-case generators will influence the results of our regression testing framework. As described in Sect. 5.1, our framework is independent of a certain generator and, hence, the applied generator is exchangeable. In this thesis, we applied a prototypical generator that performs event simulation as well as incremental depth-first search to generate covering test cases for given transition and retest test goals.

The neglection of resource factors such as the time or costs required for test-case generation as well as test-case execution may be threats. In general, testing costs are an important aspect as there are solely limited testing budgets/resources available. The selection of the test-case generator has a respective impact as varying generators require a different amount of time and further create test cases with distinct complexity. Again, our framework is independent from a certain test-case generator. In addition, the test-case execution costs could be reduced based on the application of test-suite optimization techniques such that the redundancy of (retest) test goal coverage is tackled. Therefore, it is reasonable to combine our framework with optimization techniques such as the work of Baller et al. [BLL+14; Bal17], where the coverage, costs, and profits are taken into account for the optimization. This potential combination is out of scope of this thesis and represents an open field for future research.

The type as well as the creation of artificial faults to investigate the effectiveness of our retest test selection is a potential threat. Unfortunately, we are unaware of real faults for the three evolving subject SPLs. To cope with this threat, we simulate potential faults, where we focus on erroneous artifact interactions as fault type. We focus on this fault type as our framework selects test cases for retesting already tested behavior which is potentially erroneously influenced by changes between subsequently tested variants and versions of variants. Therefore, each fault represents a simulated erroneous artifact interaction between two transitions, which is caused by changes and their impact on common behavior. In addition, we select fault sets randomly to derive varying data sets for the test of a variant or version of a variant to obtain meaningful results. However, we have to substantiate our results regarding the fault detection rate by performing more experiments with evolving delta-oriented SPLs that have a real fault history. The incorporation of different fault types is also conceivable such that model-based mutation testing for state machines [ABJ+15; LS14] becomes applicable to to simulate faults. Based on additional experiments with different fault types, we are able to reason about for which types of faults the effectiveness of our framework decreases.

For the reproducibility of our evaluation and the obtained results, we provide the prototype of our testing framework and all data gathered during the controlled experiment online.[5]

[5] `https://github.com/SLity/mbtSPLregression`

5.5 Related Work

In this section, we discuss related work regarding (1) regression-based SPL testing, (2) SPL product prioritization, and (3) model-based regression testing for single-software systems. We omit the discussion about pure model-based SPL testing approaches mainly applied for (1) test-case design and generation [RKP+05; RMP07; WSS08; SOM10; LG10; COL+11; Oli08; Far10; WGA+13; DCP+12; DPL+14; DPS14; LBL+14; LSB+17; LGS+19; LRB+19], (2) sampling [AKT+16a; PSK+10; OMR10; OZL+11; OZM+11; POS+12; LOG+12; RBR+15; DPL+15], or (3) formal SPL conformance testing [LK12; Loc13; LML15; BM16]. Those approaches facilitate the reduction of the overall effort for SPL testing by reducing the number of variants to be tested or by exploiting the explicit knowledge about commonality and variability for efficient test-case derivation and for reusing test cases during the SPL test process. Compared to our model-based SPL regression testing framework, we also apply model-based testing for the creation of test artifacts, but we further apply retest test selection to reduce the testing effort such that we allow for the reuse of test artifacts and, in addition, of test results during SPL testing. Thus, our framework focuses on a different testing strategy that is orthogonal to those existing model-based SPL testing techniques to allow for efficient SPL testing.

Furthermore, except Lochau et al. [LRB+19], none of the model-based SPL testing techniques take SPL evolution into account. Lochau et al. [LRB+19] incorporate techniques for round-trip engineering (1) to capture and update the behavioral specification of evolving SPLs in an 175% state machine test model such that version-specific 150% test models are merged in the 175% model and (2) to facilitate the co-evolution and generation of test cases based on the 175% state machine test model. Again, their technique applies test-case generation for evolving SPLs, whereas our framework focus on retest test selection and, hence, on a different strategy to reduce the testing effort. For an overview on model-based SPL testing, we refer to Oster et al. [OWE+11]. In addition, we refer to surveys on SPL testing [TTK04; ER11; OWE+11; dMdCM+11; dCMC+14; LKL12] and single-software regression testing [YH12; AET+19; ERS10; KJM+17; HO08; BMS+11] for respective overviews in general.

5.5.1 Regression-Based SPL Testing

In the context of SPLs, the concept of regression testing is adapted for (1) test process planning and managing in the industrial context [Eng10a; Eng10b; RE12a; RE12b], (2) sample-based testing [QCW07; QCR08], and (3) incremental SPL testing [dMdCC+10; UGK+08; BL14; VBM15; DFG+17; LLL+14; LLL+15; DSL+13]. We focus on the discussion regarding the incremental testing techniques as our framework also follows an incremental strategy for efficient testing of variants and versions of variants. However, none of those incremental techniques take SPL evolution and respective testing processes of versions of variants into account. Hence, our framework represents the first technique allowing for the application of regression testing by means of automated retest test selection for efficient testing of individual SPL versions and subsequent SPL versions in an incremental way.

Neto et al. [dMdCC+10; dMdCC+12; dM S10] proposed a regression testing technique for SPL integration testing. Based on the manual examination of a set of performed modifications, a test architect identifies the respective impact on variable architectural artifacts such as the source code, behavioral models, or structural models. The results are then used to manually select and prioritize test cases for retesting of an SPL architecture and variant-specific architectures. Compared to our regression testing framework, their technique combines retest test selection and prioritization on the integration testing level guided by change impact analysis. However, they perform those

steps manually, whereas our change impact analysis as well as retest test selection is automated and applied on the component testing level.

Uzuncaova et al. [UGK+08; BUK10] presented one of the first incremental SPL testing techniques. They perform incremental test-case generation based on a refinement strategy, when stepping to the next variant to be tested. For the refinement of a test case of the previous variant under test, they incorporate the features that are newly composed into the next variant to refine, i.e., regenerate the test case using Alloy. Compared to **Uzuncaova et al.** [UGK+08; BUK10], our framework performs retest test selection and abstract from the test-case generation process. In addition, our framework as well as the incremental test-case generation exploits the differences between variants under test.

Baller et al. [BL14; Bal17] proposed incremental test-case generation based on an 150% state machine test model representing the behavioral specification of an SPL and test goal profits. In each iteration of the generation, a new test goal with the highest profit is selected to generate a covering test case. During the generation, the feature annotations traversed on the test-case-specific state machine path are collected and combined as propositional formula to specify the subset of variants in a symbolical way. They reuse the generated test cases for test goals and the information on the subsets of variants on which test cases are executable to incrementally generate further test cases until all test goals are covered on all variants or a profit bound is reached. Compared to **Baller et al.** [BL14; Bal17], again, our framework performs retest test selection and abstracts from the test-case generation process. In addition, our framework is based on delta-oriented test modeling facilitating the reasoning about differences between subsequent variants under test, whereas their technique exploits the annotative state machine representation to allow for an efficient test-case generation.

The most regression-based SPL testing techniques apply delta modeling [CHS15; Sch10] as test-modeling formalism. This is reasonable as regression testing strategies exploit the information about differences [YH12] which is explicitly specified by means of (regression) deltas. In the following, we discuss delta-oriented SPL testing techniques. Compared to those techniques, we also apply delta-oriented test modeling and further adapt delta modeling for the realization of our change impact analyses for model-based regression testing of evolving SPLs.

Lochau et al. [LSK+12; LLL+14; Loc13] proposed the first technique for delta-oriented SPL testing on the component and integration testing level. Based on the adaptation of delta modeling for state machine and architectural test models, they defined a workflow for incremental SPL testing, where test artifacts and test results are reused for subsequently tested variants. When stepping to the next variant, the regression delta between the subsequent variants is used to adapt the set of test artifacts and afterwards a manual change impact analysis and retest test selection is performed. Our framework is based on their incremental workflow, especially for testing the initial SPL version. However, in contrast to **Lochau et al.** [LSK+12; LLL+14; Loc13], where retest decisions were made manually, we automated this process by applying delta-oriented change impact analysis. Furthermore, we extend the workflow for regression testing of subsequent SPL versions under test such that the reusability of test artifacts and test results is also exploited for efficient testing of versions of variants.

Lachmann et al. [LLL+15; Lac17] presented test-case prioritization for incremental SPL integration testing that adapts the delta-oriented testing approach of **Lochau et al.** [LLL+14]. They follow the same incremental workflow, where test artifacts are adapted based on the incorporation of architecture regression deltas. They further exploit the change operations between subsequently tested variant-specific architectures to prioritize reusable test cases. By examining the changes covered

by a reusable test case, each test case gets a weight incorporated in the prioritization. Reusable test cases providing a high weight are to be reexecuted first on the next variant under test. **Lachmann et al.** [LLA+16] extended their work by taking also the changes to the internal component behavior into account for a more fine-grained analysis and prioritization. In addition, they examined the dissimilarity between test cases to increase the early coverage of changes to be retested. **Lachmann et al.** [LBL+17] integrated risk analysis into their prioritization technique such that weights of reusable test cases to be ordered are derived by examining the failure impact as well as failure probability of features and components of a variant under test. Compared to **Lachmann et al.** [LLL+15; Lac17], our framework is applied on the component testing level and reasons about differences on delta-oriented state machine test models. We further perform retest test selection, whereas their technique prioritizes reusable test cases for retesting. However, the extension of their work [LLA+16], where the internal component behavior is incorporated, could benefit from our test-modeling formalism and delta-oriented change impact analysis by means of incremental model slicing.

Dukaczewski et al. [DSL+13] adapted the testing approach of **Lochau et al.** [LLL+14] to allow for delta-oriented requirement-based SPL testing on the system testing level. They apply delta modeling for textual requirement specifications and associated test cases. As change impact analysis is hard to perform on requirements written in natural language, they uses four strategies to select reusable test cases to be retested depending on the available development and test artifacts, namely random-based, meta-data-based, history-based, or model-based selection. In contrast to **Dukaczewski et al.** [DSL+13], our testing framework is defined for the component testing level. However, both techniques perform retest test selection, but compared to **Dukaczewski et al.** [DSL+13], we exploit the results of our automated change impact analyses to control the selection process.

Varshosaz et al. [VBM15] proposed delta-oriented test-case generation based on finite state machine test models. A test model is encoded as DeltaJava program based on the adaptation of delta-oriented programming. Their technique abstracts from remove operations and focuses on modify and add operations, where they exploit the incremental structure for the incremental generation of test cases based on existing ones. In contrast to **Varshosaz et al.** [VBM15], our framework performs retest test selection and abstract from test-case generation. Compared to our test-modeling formalism, their technique is also based on state machine test models, but they use a different state machine dialect and further encode delta-oriented state machines as DeltaJava program.

Damiani et al. [DGT13; DFG+17] presented a technique for delta-oriented model-based testing of Java programs. They also exploit delta-oriented programming for capturing the commonality and variability of specifications, but they adapt the concept for tabular program specifications. In addition, they extend the tool FineFit used to derive test cases from tabular program specifications, to handle the change operations applied to the tabular specification when stepping to the next variant to be tested. In contrast to **Uzuncaova et al.** [UGK+08; BUK10] as well as **Varshosaz et al.** [VBM15], their technique also incorporates remove operations during incremental test-case generation. In contrast to **Damiani et al.** [DGT13; DFG+17], we perform retest test selection and further use delta-oriented state machines as behavioral specification.

5.5.2 SPL Product Prioritization

In the literature, several approaches for product prioritization exists (1) examining the similarity of feature configurations [ATM+14; SSR14], (2) incorporating domain knowledge [DPC+17; EBA+11;

JHF+12; LJC+14], or (3) by applying multi-objective optimization [PSS+16; HPP+14; DPL+16; BLL+14]. We discuss those techniques w.r.t. our prioritization technique (cf. Sect. 5.3) in the following.

Feature Configuration Similarity. Similarity-based prioritization techniques determine testing orders, where subsequent variants to be tested are very dissimilar to all preceding ones to increase the coverage of, e.g., pairwise feature interactions. In contrast, our prioritization technique focuses on similar variants such that we identify a sequence of variants to be tested, where the total number of differences between subsequent variants gets minimized in order to increase the reuse potential during regression testing of an SPL version in time.

Al-Hajjaji et al. [ATM+14; ATL+16; Al-17; AKS+17] proposed a product prioritization approach, where the Hamming distance is applied as distance metric to reason about the similarity of feature configurations. Based on a certain variant used as starting point, i.e., the all-yes-config which comprises the largest number of features in its feature configuration, their technique always selects the most dissimilar variant to all previously integrated variants. **Al-Hajjaji et al.** [ALL+17] extended their work to also allow for a prioritization based on solution space artifacts, where they use delta modeling as variability implementation technique. The extension further facilitates the combination of problem as well as solution space information to perform the prioritization. Compared to our approach, their technique is mainly applied for feature configurations and, hence, in the problem space. In addition, we encode the optimization problem as TSP and adapt different existing graph heuristics to obtain an approximately optimal solution, whereas their technique applies a greedy-based heuristic. However, their extension incorporating deltas to examine the similarity is very similar to our approach. **Al-Hajjaji et al.** [ALL+17] use the hamming distance to determine the similarity between variant-specific delta sets, whereas we directly exploit the regression delta between arbitrary variants capturing the respective differences.

Sánchez et al. [SSR14] presented five different criteria by means of feature model metrics for product prioritization. Those criteria incorporate, e.g., the complexity as well as similarity of variants and result in different testing orders. However, all criteria are defined to maximize the rate of early fault detection, where the applied criterion sets the focus on variants to be tested first, e.g., the most complex ones. In contrast to our approach, their technique solely prioritizes variants in the problem space. In addition, independent from the criterion to be applied for prioritization, their technique selects the next variant to be integrated in the testing order greedy-based without taking the already selected variants into account.

Incorporation of Domain Knowledge. Just as similarity-based techniques, prioritization techniques that incorporate additional domain knowledge to determine testing orders also aim for the increase of the early fault detection rate by selecting dissimilar variants. In contrast, our prioritization technique does not require additional information as the information about differences between variants in terms of the number of change operations is captured in a regression delta. Furthermore, as already mentioned, we focus on similar subsequent variants under test to increase the reuse potential during SPL regression testing.

Devroey et al. [DPC+13; DPC+17] combined behavioral as well as statistical analyses for product prioritization. Based on a featured transition system capturing the behavior and a usage model comprising the behavioral usage scenarios and their execution probabilities of an SPL under test, the technique always selects the next variant that covers the behavior with the highest execution probability. Hence, those variants are to be tested first which have a high priority w.r.t. the usage

scenarios captured in the usage model. Compared to **Devroey et al.** [DPC+13; DPC+17], our prioritization technique also takes the behavior of an SPL into account, but we focus solely on the differences between variants and the minimization of the overall number of differences.

Ensan et al. [EBA+11] proposed a combined sampling and prioritization technique. They exploit a goal model capturing preferences and objectives of stakeholders as well as a mapping of goals to features of a feature model to control their technique. First, a domain expert selects a (sub)set of important features to achieve stakeholders goals to obtain a sample of the variant set. Second, the coverage by means of goal satisfaction of a feature configuration is utilized to prioritize the computed sample. Therefore, variants are to be tested first which comprise the most desirable features in their respective configurations w.r.t. the decisions made by the domain experts. Compared to **Ensan et al.** [EBA+11], our technique has a different objective for the computation of testing orders, i.e., the increase of the reuse potential during SPL regression testing. In addition, our technique do not focus on the sampling of the variant set of the SPL under test, but our prioritization is combinable with sampling strategies [VAT+18] which can be applied in advance.

Johansen et al. [JHF+12] presented a sampling technique that implicitly provides a prioritization of variants. They derive weights for features by incorporating the information how often a feature has been sold in certain variants. Based on a feature model and those weights, the sampling algorithm ICPL [JHF12] selects variants to be integrated in the resulting sample that have a high coverage of, e.g., pairwise feature interactions and also provide a high weight w.r.t. the features contained in their respective feature configurations. Due to the incremental coverage-driven selection, the resulting sample is prioritized such that the important variants with high sales numbers are tested first. **Lopez-Herrejon et al.** [LJC+14] adapted the weighting scheme of **Johansen et al.** [JHF+12], i.e., the incorporation of sales numbers of features, and also proposed a combined technique for prioritized samples using an evolutionary algorithm called parallel prioritized product line genetic solver. Compared to both techniques, again, our technique follows a different objective for the computation of testing orders and we do not focus on sampling, but can incorporate its application.

Multi-Objective Optimization. Based on the application of search-based techniques, testing orders of variants are also computed such that several objectives such as costs, coverage, dissimilarity etc. are fulfilled at the same time. In contrast to those mulit-objective optimization techniques, we focus solely on one objective, i.e., the minimization of the total number of differences between consecutively tested variants. However, for the computation of such an optimized testing order encoded as TSP, the application of search-based techniques and the incorporation of further objectives is also possible.

Parejo et al. [PSS+16] proposed seven distinct objective functions based on the combination of functional, e.g., feature interaction coverage, and non-functional properties, e.g., fault history, to facilitate a multi-objective prioritization. For solving the optimization problem, they apply the evolutionary algorithm NSGA-II. Compared to **Parejo et al.** [PSS+16], we focus on optimized testing orders that increase the exploitable reuse potentials between subsequently tested variants, where we take their similarity by means of differences captured as change operations in regression deltas into account.

Henard et al. [HPP+14] presented a search-based technique that computes a prioritized sample w.r.t. the objectives of maximizing the feature interaction coverage, minimizing the test-suite size, and minimizing the testing costs. For the prioritization, they defined two techniques, namely the

local maximum distance (LMD) and the global maximum distance (GMD) prioritization. The LMD prioritization selects in each step a pair of variants from the determined sample to be integrated in the testing order, where the variants have the highest dissimilarity value w.r.t. their feature configurations determined based on the Jaccard metric. For this technique, they do not take already selected variants into account. In contrast, the GMD prioritization always selects and integrates the next variant from the sample that provides the highest dissimilarity value regarding all previously selected variants in the testing order. Compared to **Henard et al.** [HPP+14], the FARIN insertion heuristic which we adapt for our approach follows a similar variant selection as the GMD prioritization, but they differ in the integration of a variant into the testing order. For the GMD prioritization, a selected variant is added to the end of the testing order, whereas for the FARIN insertion heuristic, a selected variant can be added at any position of the order to provide the minimal increase of the overall number of differences to be minimized.

Devroey et al. [DPL+16] proposed another multi-objective prioritization technique, where they exploit the behavioral analysis of their previous work [DPC+13]. Based on a featured transition system capturing the behavior of an SPL under test, the technique prioritizes variants w.r.t. the coverage of pairwise feature interactions as well as SPL behavior. Hence, they always select the most dissimilar variants to increase the early fulfillment of their coverage objectives. Compared to **Devroey et al.** [DPL+16], we also take the behavior of an SPL into account, but reason about the similarity of variants by means of differences captured as change operations in regression deltas, whereas their technique is coverage-driven.

Baller et al. [BLL+14; Bal17] realized a framework for test-suite optimization, where as a side result a testing order is provided. They incorporate the mapping between test goals, test cases, and variants as well as respective costs and profits to find an SPL test suite with minimized costs and maximized profits. To compute an approximately optimal solution, a heuristic is defined, where dissimilar variants are selected incrementally to facilitate the early fulfillment of the cost and profits objectives. In contrast to our approach, the complete mapping between test goals, test cases, and variants as well as the respective costs and profits have to be known in advance before their technique is applicable. Furthermore, **Baller et al.** [BLL+14; Bal17] optimize SPL test suites, whereas we prioritize variants to increase the reuse potential during SPL regression testing.

5.5.3 Model-Based Regression Testing

For single-software systems, different techniques that combine the benefits of model-based testing and regression testing for efficient software testing have been proposed. In this section, we focus on the discussion of model-based regression testing techniques that, similar to our model-based SPL regression testing framework, apply behavioral UML models as test models. In addition, we refer to Sect. 4.4.4 for a discussion regarding model-based regression testing techniques that use program or model slicing as change impact analysis. For a general overview on UML-based regression testing techniques, we refer to **Fahad and Nadeem** [FN08].

Existing techniques for model-based regression testing mainly apply sequence diagrams [BLS02; NZR09; ANI+07; MTN11] or activity diagrams [CPS02] as test-modeling formalism, whereas our framework is based on state machines. **Briand et al.** [BLS02; BLH09] proposed UML-based retest test selection, where the original as well as modified UML models, i.e., class and sequence diagrams, are compared to identify changes and their impact. Based on the existing traceability of test cases

to sequence diagrams and their contained messages, test cases are categorized as retestable if a test case is mapped to a modified message. **Naslavsky et al.** [NZR09; NZR10] also exploit the traceability between test cases and sequence diagrams established during test-case generation. A test case is selected for a retest if it is mapped to a modified or modification-influenced element, e.g., message, of a sequence diagram, where the change impact analysis is performed via model differencing. Compared to our retest test selection, those techniques apply model differencing as change impact analysis resulting in different retest decisions to be made as our framework applies incremental model slicing to identify changes in the execution dependencies of a transition. In addition, our technique is guided based on our retest test coverage criterion, whereas those techniques rely on existing traceability or mapping information.

In contrast to model differencing, the technique of **Rothermel and Harrold** [RH94], where the original and modified control flow graph are traversed in parallel to identify changes and their impact, is adapted for UML models. **Ali et al.** [ANI+07] presented a retest test selection approach based on extended concurrent control flow graphs which are generated from class and sequence diagrams of the SUT. Test cases are selected for retesting if they are mapped to modified or modification-affected control flow graph elements. **Mansour et al.** [MTN11] detected modifications and their impact in UML interaction overview diagrams which, in turn, use sequence diagrams to specify interactions between classes of the system under test. Test cases that contain modified method calls are selected for reexecution. **Chen et al.** [CPS02] applied activity diagrams for the behavioral specification of a system under test and select two types of test cases to be retested. The selection of targeted tests is guided based on modified or modification-affected activity diagram elements. In addition, they use a risk analysis to select safety tests to be reexecuted. Compared to our retest test selection, those techniques select test cases that traverse the modified or modification-affected UML model elements, whereas our selection is guided based on the retest test coverage criterion.

There are also techniques that apply state machine test modeling to perform model-based regression testing [FIM+07; Muc07]. **Farooq et al.** [FIM+07; FIM+10] presented an approach and its tool support Start for selective regression testing, where changes to class diagrams and state machines are used to select test cases for reexecution. They incorporate both UML models to reason about different types of changes, namely class- and state-driven changes. By comparing the original and the modified versions of the UML models, they identify modified transitions and select those test cases for a retest which are mapped to those modified transitions. **Muccini** [Muc07] proposed a retest test selection approach, where model differencing is applied to state machines specifying the internal behavior of components to facilitate change impact analysis. Test cases are selected based on their simulation on the determined difference model. In case a test case traverses modified elements in the difference model, the test case is selected for retesting. Again, compared to our retest test selection, both techniques apply model differencing as change impact analysis resulting in different retest decision to be made as our framework is guided by retest test goals to be covered. Furthermore, those state machine test models have to be flattened, whereas our test-modeling formalism takes hierarchy and concurrency of state machines into account.

5.6 Summary

The application of retest test selection is a well-known strategy in regression testing to reduce the set of test cases to be reexecuted in order to validate that already tested behavior is not erroneously

influenced by changes [YH12]. In the context of evolving SPLs, the adoption of retest test selection allows for tackling the potential redundancy during SPL testing introduced by the shared commonality. Therefore, we proposed a model-based regression testing framework that unites our delta-oriented test-modeling formalism and the delta-oriented change impact analyses to facilitate retest test selection for efficient and effective testing of variants and versions of variants. The framework incrementally tests consecutive SPL versions by exploiting the reuse potential of test artifacts and test results obtained based on the testing processes of preceding SPL versions such that a reduction of the overall testing effort is achievable. To guide the retest test selection, we defined a retest test coverage criterion that incorporates the results of the slicing-based change impact analysis to derive retest test goals. Hence, we select reusable test cases and further generate retest test cases to ensure retest coverage during the regression testing of variants and versions of variants. Furthermore, we proposed a prioritization technique for computing reuse-optimized testing orders, where the similarity between subsequent variants to be tested is taken into account such that our slicing-based impact analysis and the followed retest test selection benefit for the incremental testing of consecutive variants under test. For the comparison of variants regarding their similarity, we exploit the explicit knowledge about differences specified by our delta-oriented test-modeling formalism.

We prototypically implemented our model-based SPL regression testing framework and applied it to the three evolving model-based SPLs (cf. Sect. 3.4) to evaluate the efficiency and effectiveness of our framework. The results show that our framework reduces the number of test cases to be reexecuted for consecutively testing variants and versions of variants compared to the retest-all strategy. The reduced retest test suite ensures retest coverage based on the explicit generation of retest test cases. Furthermore, our framework achieves a good fault detection rate w.r.t. erroneous artifact interactions even with smaller retest test suites compared to retest-all. The preliminary examination of the influence of varying testing orders on our framework also provides positive results. In contrast to dissimilarity-based testing orders [ALL+17; ATM+14; HPP+14], reuse-optimized testing orders allow for less retest decisions to reason about and, therefore, in smaller retest test suites for regression testing of subsequent variants under test.

6 Conclusion

Testing is a crucial and challenging activity for the successful development of (complex) software systems [Har00; SLS11; AO16] and gets even more challenging in the context of SPLs as variability introduces an additional dimension of complexity [McG10; ER11; OWE+11; dCMC+14; LKL12]. An SPL is completely tested by testing each realizable variant, where often the number of variants grows exponentially in the number of features. Besides the vast number of potential variants to be tested, the inherent commonality shared between them leads to redundant testing processes as reusable test cases are executed more than once to validate the same functionality for different variants again. The variability and the testing redundancy impede the practical application of single-software testing techniques, where each variant would be tested individually without taking the explicit knowledge about the shared commonality and variability into account [TTK04; MI07; McG10; OMR10; ER11; LKL12]. Furthermore, due to the increasing longevity of software systems, their development has to face software evolution [Leh96; SB99; GG08; MSC14; BP14]. SPL testing has to cope with the evolution of reusable software artifacts and their interdependencies resulting in an even more challenging quality assurance activity as typically not only a single system is influenced by changes of an evolution step, but rather a potentially large set of variants. Hence, quality assurance has also to be ensured after SPL evolution by testing respective versions of variants [RE12b; RE12a]. However, there is a lack of SPL testing techniques that handle both, the variability and the evolution of an SPL in order to facilitate the efficient testing of variants and versions of variants based on the exploitation of their shared commonality and variability.

6.1 Discussion

In this thesis, we tackled the challenges of the potential testing redundancy by means of redundant test-case executions as well as of the quality assurance after SPL evolution. We proposed a framework for model-based regression testing of variants and versions of variants and, therefore, efficient incremental testing of evolving SPLs. The framework combines model-based testing [UL06; UPL12] and retest test selection as regression testing strategy [YH12] which are both well-suited for SPL testing [Oli08; Loc13; Eng10b; McG10] in order to facilitate (1) the automated generation of test cases [UL06; UPL12] reusable between consecutively tested variants and versions of variants, and (2) the reduction of test cases to be (re)executed for validating that already tested behavior shared between variants and versions of variants is not erroneously influenced by their differences.

For the definition of the testing framework, three crucial activities had to be specified by incorporating the dimensions of variability and evolution of an SPL. Those activities are the process of test modeling, the application of change impact analyses, and the process of retest test selection for variants and versions of variants. In the following, we recall the issues which arised for the specification of the activities and discuss the solutions based on our contributions and observations. In addition, we provide insights for each activity, how the respective solution can be applied for the development of evolving SPLs in general.

Test Modeling for Variants and Versions of Variants. Test modeling is fundamental for the successful application of model-based testing [UL06; UPL12]. As motivated in Chapt. 1 and in Chapt. 3, in the context of evolving SPLs, a test-modeling formalism has (1) to handle both, variability and evolution in the same way, (2) to be adaptable and, thus, applicable for various test-model artifact types, (3) to document the complete evolution history, and (4) to facilitate the automated analysis about the evolution impact. However, existing techniques for model-based SPL testing [Loc13; COL+11; LLL+14; DPL+14; VBM15; LLL+15; DFG+17; OWE+11] as well as for managing SPL evolution in the solution space [ST00; ALR+05; AMC+07; DGR+10; SSA13a; HRR+12; LSK+13; KLL+14; NBA+15] have at least one limitation w.r.t. those requirements such that adequate test modeling for variants and versions of variants was an open issue.

We tackled this issue and addressed the four requirements by proposing higher-order delta modeling as transformational variability implementation technique to capture the variable behavioral specification of variants and versions of variants by the same means. A variable test model of an SPL version is represented by a delta model, where the differences between variants are explicitly captured as transformations encapsulated in deltas. For the evolution of an SPL, we transform version-specific delta test models by altering their encapsulated delta set via additions, removals, and modifications of deltas specified via higher-order deltas. Therefore, a higher-order delta test model captures the complete evolution history of an SPL. Although we instantiated higher-order delta modeling for state machines, the modeling technique is adaptable for different types of development artifacts. Furthermore, the explicit knowledge about the commonality and difference between subsequently tested variants and versions of variants by means of deltas facilitates change impact analysis as well as retest test selection as both concepts exploit the commonality and focus on the differences during their application.

In the context of SPL testing, higher-order delta modeling facilitates the instantiation for additional test model artifact types and, therefore, can support model-based testing of evoling SPLs also on other testing levels. For instance, by instantiating our modeling technique for software architectures, incremental SPL integration testing [LLL+14; LLL+15; LLA+16] can be adapted for model-based integration testing of variants and versions of variants. Furthermore, higher-order delta modeling is not restricted to the application domain of software testing, but can be applied as variability implementation technique in SPLE for evolving SPLs in general. However, for a general application, an adequate tool support is much needed as the complexity of a higher-order delta model increases w.r.t. the dimensions of variability and evolution. In its current version, our prototypical implementation is not sufficient and has to be improved, e.g., based on the definition of views etc., to provide an adequate support for an SPL domain engineer.

Change Impact Analysis for Variants and Versions of Variants. Change impact analysis is essential for successful regression testing [YH12]. As motivated in Chapt. 1 and in Chapt. 4, in the context of evolving SPLs, impact analysis has to facilitate (1) the detection of (test-model) changes to already tested behavior between subsequently tested variants and version of variants, and (2) the determination of the impact of an evolution step to the set of variants on the solution space level in terms of new, removed, unchanged, or modified variants. However, existing techniques for change impact analysis in the context of single-software regression testing [Boh96; Arn96; Leh11b; Leh11a; Bin98; YH12] or SPL evolution management [NSS16; SBT16; TBK09; BKL+16] cannot be applied to support incremental testing of evolving SPLs as variability is not taken into account or solely problem space

evolution is analyzed. Therefore, change impact analysis for regression testing of evolving SPLs was an open issue.

We tackled this issue by proposing two techniques for incremental change impact analysis. First, we introduced incremental model slicing for guiding retest test selection during incremental testing of variants and versions of variants. By exploiting the shared commonality and the differences between subsequently tested variants and versions of variants by means of deltas, our slicing technique automatically identifies changed execution dependencies. Changed execution dependencies indicate behavior potentially influenced by changes to the test model between subsequent variants and also between modified versions of variants and, therefore, refer to behavior to be retested. Second, we defined an incremental delta set derivation allowing for the reasoning about higher-order delta applications. Higher-order deltas specify how the delta set of a version-specific delta model changes in terms of additions, removals, and modifications of deltas. The incremental delta set derivation exploits those changes to infer and reason about the respective changes on variant-specific delta sets. The reasoning process results in a categorization of how the variant set alters between consecutively tested SPL versions in terms of added, modified, and unchanged versions of variants which is exploited to guide the incremental test process for evolving SPLs.

Both analysis techniques are currently applied in the context of regression testing of evolving SPLs, but are also applicable for other tasks in SPLE in general. On the one hand, our slicing technique can also be used to support model comprehension which is a typical application scenario of slicing techniques [ACH+13]. Based on the exploitation of delta modeling, incremental slicing is also adaptable for program slicing, where a respective engineering effort is required to adapt the incremental concepts to delta-oriented programming [SBB+10]. On the other hand, our incremental delta set derivation can also be applied to support the planning of SPL evolution. Based on the identification which and how variants are affected based on changes to the delta model, we are able to estimate the potential effort to apply the planned evolution step. Furthermore, the combination of higher-order delta modeling and temporal feature modeling [NSS16; NES17] would allow for comprehensive change impact analysis of both, problem as well as solution space evolution.

Retest Test Selection for Variants and Versions of Variants. Retest test selection facilitates the reduction of redundant test case executions [YH12; KJM+17]. As motivated in Chapt. 1 and in Chapt. 5, in the context of evolving SPLs, retest test selection allows for tackling the potential redundancy during SPL testing arising based on the shared commonality between variants and versions of variants by selecting reusable test cases for their reexecution. For the selection of reusable test cases to be retested, the differences between subsequently tested variants or versions of variants and their impact to shared, yet already tested behavior has to be taken into account based on the application of change impact analysis. However, existing SPL retest test selection techniques [dMdCC+10; LLL+14; DSL+13] either perform a manual selection or do not incorporate automated change impact analysis to support the selection process and, furthermore, do not take the evolution of SPLs into account. Therefore, automated retest test selection guided by change impact analyses and applied for incremental testing of variants and versions of variants was an open issue.

We tackled this issue by proposing automated coverage-driven retest test selection. For the guidance of the selection process, we defined a new retest coverage criterion that incorporates the results of our slicing-based change impact analysis between consecutively tested variants as well as versions of variants. By taking the identified changed execution dependencies into account, we

derive retest test goals which are to be covered by selecting reusable test cases for their reexecution. Therefore, our selection technique allows for the validation that already tested behavior is not erroneously influenced when stepping to the next variant or version of a variant to be tested such that a reduction of the number of test cases to be executed for testing evolving SPLs is achieved.

In its current version, we defined the retest coverage criterion based on the incorporation of state machine test modeling and the results of slicing-based change impact analysis. However, the definition of the retest coverage criterion allows for a more general application of retest test selection in the context of SPL testing. By adapting the criterion to a different test-model artifact type and to the results of a respective change impact analysis, the coverage-driven retest test selection is applicable on additional testing levels, e.g., to facilitate efficient incremental integration testing [LLL+14; LLL+15; LLA+16] also for evolving SPLs. Furthermore, our retest test selection reduces the number of test cases to be reexecuted, but is restricted to the fulfillment of the coverage criterion. The set of selected test cases still can comprise redundancy by means of (retest) test goal coverage such that the set can be further reduced based on the incorporation of test-suite minimization techniques [YH12].

Efficient Model-Based Regression Testing of Variants and Versions of Variants. Based on the described contributions, we are finally able to provide an answer to the main research question which was defined as follows:

How can we efficiently test evolving software product lines based on the reduction of redundant test-case executions?

In this thesis, we proposed a framework for model-based SPL regression testing of variants and versions of variants. The framework unites the delta-oriented test-modeling formalism, the delta-oriented change impact analyses, and the coverage-driven retest test selection to facilitate efficient incremental testing of consecutive SPL versions. During the incremental test process, the framework exploits the reuse potential of test artifacts and test results of already tested variants and versions of variants to reduce the overall testing effort by tackling the potential redundancy during testing of evolving SPLs. The framework is prototypically implemented and evaluated by means of three evolving SPLs showing that it achieves a reduction of redundant test-case executions while maintaining the effectiveness by means of a good fault detection rate of erroneous artifact interactions. The resulting reduction of the overall test effort enables a more targeted use of the limited test resources and, hence, facilitates efficient quality assurance of variants and versions of variants achieved via model-based regression testing.

In the end, we believe that the obtained results, the comparison to related work as well as the fact that our framework represents the first technique that applies regression testing for efficient testing of individual SPL versions and subsequent SPL versions justify the statement of being a novel contribution in the research community.

6.2 Future Work

To further improve our current contributions and the good results by means of an achieved reduction of the overall testing effort, we envision future work. First, our contributions should be evaluated by means of additional case studies of evolving SPLs in order to facilitate the generalization of our obtained results. In the following, we discuss further potential improvements w.r.t. the three activities of our model-based regression testing framework.

Future Work for Test Modeling. As common for variability implementation techniques [SRC+12], the complexity of a variable artifact, e.g., a higher-order delta test model, represents a drawback. Variability as well as evolution introduce new dimensions of complexity to be handled by a domain engineer if the variability implementation technique is directly applied for the development of evolving SPLs. In contrast to the direct application, higher-order delta modeling is also exploitable as some kind of data structure in variation control systems [LBG17]. However, an assessment of higher-order delta modeling applied as variability implementation technique for SPLE of evolving real-world SPLs is desirable. Such an assessment could be performed similar to the work of Ferreira et al. [FGF+14] for feature-oriented, of Figueiredo et al. [FCS+08] for aspect-oriented, or of Diniz et al. [DVG+17] as well as Hamza et al. [HWE17] for delta-oriented programming. A user study reviewing the general applicability and modability of higher-order delta modeling is also conceivable in this context.

In this thesis, we apply higher-order delta modeling instantiated for state machines as test-modeling formalism allowing for model-based regression testing on the component testing level. Due to the adaptability of our modeling technique for other test-model artifact types than state machines, e.g., software architectures, we are able to instantiate the formalism, e.g., to support incremental SPL integration testing [LLL+14; LLL+15; LLA+16]. The presented tool support could also be improved, e.g., by graphical editors, to facilitate a more intuitive (test) modeling of evolving SPLs.

Another aspect to be potentially improved in future work is given in the fact that higher-order delta modeling solely captures solution space evolution. In general, the evolution of an SPL impacts both the solution as well as problem space [SB99; PBvdL05; BP14]. To facilitate a comprehensive management of evolving SPLs, the combination of higher-order delta modeling and temporal feature modeling [NSS16; NES17] is desirable. This combination further allows for more elaborated analyses to support the management of SPL evolution.

Future Work for Change Impact Analysis. For incremental model slicing, we currently focus on control dependencies as proposed by Kamischke et al. [KLB12]. The applied set of dependencies is sufficient for change impact analysis of abstract behavioral specifications as used in this thesis. However, for the application of our incremental slicing technique as a more general analysis technique in SPLE, the incorporation of data dependencies is aspired. By taking also data dependencies into account, the range of applications for model-based SPLs is increased and further facilitates the identification of the change impact for execution dependencies w.r.t. the read/write access of variables.

As already mentioned, the evaluation of our techniques by means of real-world evolving SPLs is an important aspect to substantiate our obtained results. By focusing on the application of higher-order delta modeling and how variants are generated based on the combination of deltas, we can examine the catolog of delta dependencies introduced in this thesis. Therefore, an investigation regarding the completeness of the catalog is desirable and if necessary, the catalog can be extended by newly identified delta dependencies.

Future Work for Retest Test Selection. Based on the retest test coverage criterion, our retest test selection technique selects every reusable test case for reexecution that covers at least one retest test goal. As shown in our controlled experiments (cf. Sect. 5.4), we reduce the number of reexecuted test cases compared to retest-all, but the set of selected test cases still comprises redundancy by means

of (retest) test goal coverage. Therefore, the incorporation of test-suite optimization techniques such as the minimization technique of Baller et al. [BLL+14; Bal17], where costs and profits of test artifacts are also taken into account, is desirable in order to increase the achievable reduction of the testing redundancy during incremental testing of evolving SPLs.

Furthermore, the application of our retest test selection for additional testing levels such as for integration testing is conceivable. For such an adaptation, our technique requires a respective set of reusable test cases from which test cases can be selected for reexecution. To guide the automated retest test selection, we also require a change impact analysis that identifies modification-affected parts of the system w.r.t. the testing level under consideration such that we are able to derive retest test goals accordingly.

Another aspect to be potentially investigated in future work is the influence of testing orders on incremental testing of variants and versions of variants. Existing techniques [HPP+14; ATL+16; PSS+16; LJC+14; SSR14; DPC+13; ALL+17] focuses on the early fault detection and, hence, on the dissimilarity between subsequent variants under test, whereas we focus on the similarity in order to increase the exploitable reuse potential between subsequently tested variants. As both objectives are contrary, future prioritization techniques should also focus on an optimized trade-off to facilitate a combination of efficient retest test selection and early fault detection.

Bibliography

[90] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (Dec. 1990), pp. 1–84.

[ABC+11] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2011.

[ABC+13] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. J. Li, and H. Zhu. "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation". In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001.

[ABJ+15] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. "Killing Strategies for Model-Based Mutation Testing". In: *Softw. Test. Verif. Reliab.* 25.8 (Dec. 2015), pp. 716–748.

[ABK+16] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2016.

[ABT+16] M. Al-Hajjaji, F. Benduhn, T. Thüm, T. Leich, and G. Saake. "Mutation Operators for Preprocessor-Based Variability". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 81–88.

[ACH+13] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt. "State-based Model Slicing: A Survey". In: *ACM Comput. Surv.* 45.4 (2013), 53:1–53:36.

[ACL+11a] M. Acher, P. Collet, P. Lahire, and R. B. France. "Decomposing Feature Models: Language, Environment, and Applications". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Nov. 2011, pp. 600–603.

[ACL+11b] M. Acher, P. Collet, P. Lahire, and R. B. France. "A Domain-Specific Language for Managing Feature Models". In: *Proceedings of the ACM Symposium on Applied Computing*. SAC '11. TaiChung, Taiwan: ACM, 2011, pp. 1333–1340.

[ACL+11c] M. Acher, P. Collet, P. Lahire, and R. B. France. "Slicing Feature Models". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 424–427.

[AET+19] N. b. Ali, E. Engström, M. Taromirad, M. R. Mousavi, N. M. Minhas, D. Helgesson, S. Kunze, and M. Varshosaz. "On the Search for Industry-Relevant Regression Testing Research". In: *Empirical Software Engineering* (Feb. 2019).

[AGP+15] F. Angerer, A. Grimmer, H. Prähofer, and P. Grünbacher. "Configuration-Aware Change Impact Analysis". In: *IEEE/ACM International Conference on Automated Software Engineering*. Nov. 2015, pp. 385–395.

[AGV15] P. Arcaini, A. Gargantini, and P. Vavassori. "Generating Tests for Detecting Faults in Feature Models". In: *IEEE International Conference on Software Testing, Verification and Validation*. 2015, pp. 1–10.

[AHK+93] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. "Incremental Regression Testing". In: *International Conference on Software Maintenance*. Sept. 1993, pp. 348–357.

[AKE12] W. Abdelmoez, H. Khater, and N. El-Shoafy. "Comparing Maintainability Evolution of Object-Oriented and Aspect-Oriented Software Product Lines". In: *International Conference on Informatics and Systems (INFOS)*. May 2012, SE-53-SE–60.

[AKS+17] M. Al-Hajjaji, J. Krüger, S. Schulze, T. Leich, and G. Saake. "Efficient Product-Line Testing Using Cluster-Based Product Prioritization". In: *IEEE/ACM International Workshop on Automation of Software Testing*. May 2017, pp. 16–22.

[AKT+16a] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. "IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling". In: *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. 2016, pp. 144–155.

[AKT+16b] S. Ananieva, M. Kowal, T. Thüm, and I. Schaefer. "Implicit Constraints in Partial Feature Models". In: *Proceedings of the International Workshop on Feature-Oriented Software Development*. FOSD 2016. Amsterdam, Netherlands: ACM, 2016, pp. 18–27.

[Al-17] M. Z. S. Al-Hajjaji. "Similarity-Driven Prioritization and Sampling for Product-Line Testing". PhD thesis. Otto-von-Guericke-Universität Magdeburg, 2017.

[ALL+17] M. Al-Hajjaji, S. Lity, R. Lachmann, T. Thüm, I. Schaefer, and G. Saake. "Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing". In: *International Workshop on Variability and Complexity in Software Design (VACE@ICSE)*. May 2017, pp. 34–40.

[ALR+05] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. "Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution". In: *RAM-SE'05-ECOOP'05 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Proceedings, Glasgow, UK, July 15, 2005*. 2005, pp. 3–16.

[AMC+05] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho. "Extracting and Evolving Mobile Games Product Lines". In: *Proceedings of the International Conference on Software Product Lines*. SPLC'05. Rennes, France: Springer-Verlag, 2005, pp. 70–81.

[AMC+07] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. "Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming". In: *Transactions on Aspect-Oriented Software Development IV*. Ed. by A. Rashid and M. Aksit. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 117–142.

[Ang14] F. Angerer. "Variability-Aware Change Impact Analysis of Multi-Language Product Lines". In: *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 903–906.

[ANI+07] A. Ali, A. Nadeem, M. Z. Z. Iqbal, and M. Usman. "Regression Testing Based on UML Design Models". In: *Pacific Rim International Symposium on Dependable Computing*. Dec. 2007, pp. 85–88.

[AO16] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[APG16] F. Angerer, H. Prähofer, and P. Grünbacher. "Modular Change Impact Analysis for Configurable Software". In: *IEEE International Conference on Software Maintenance and Evolution*. Oct. 2016, pp. 468–472.

[AR11] M. Acharya and B. Robinson. "Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems". In: *Proceedings of the International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 746–755.

[Arn96] R. S. Arnold. *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[AS12] M. Aggarwal and S. Sabharwal. "Test Case Generation from UML State Machine Diagram: A Survey". In: *International Conference on Computer and Communication Technology*. Nov. 2012, pp. 133–140.

[ATK17] N. Almasri, L. Tahat, and B. Korel. "Toward Automatically Quantifying the Impact of a Change in Systems". In: *Software Quality Journal* 25.3 (Sept. 2017), pp. 601–640.

[ATL+16] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake. "Effective Product-Line Testing Using Similarity-Based Product Prioritization". In: *Software & Systems Modeling* (Dec. 2016).

[ATM+14] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. "Similarity-Based Prioritization in Software Product-line Testing". In: *Proceedings of the International Software Product Line Conference*. SPLC '14. Florence, Italy: ACM, 2014, pp. 197–206.

[Bal17] H. Baller. "Multikriterielle Minimierung und Priorisierung von Test-Suiten für Software-Produktlinien". PhD thesis. Braunschweig University of Technology, Germany, 2017.

[Bat05] D. Batory. "Feature Models, Grammars, and Propositional Formulas". In: *Proceedings of the International Software Product Line Conference*. Ed. by H. Obbink and K. Pohl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 7–20.

[BDG+06] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel. "A Formalisation of the Relationship between Forms of Program Slicing". In: *Science of Computer Programming* 62.3 (2006). Special issue on Source code analysis and manipulation (SCAM 2005), pp. 228–252.

[BF14] P. Bourque and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014.

[BH04] D. Binkley and M. Harman. "A Survey of Empirical Results on Program Slicing". In: *Advances in Computers* 62.105178 (2004), pp. 105–178.

[BH93] S. Bates and S. Horwitz. "Incremental Program Testing Using Program Dependence Graphs". In: *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. Charleston, South Carolina, USA: ACM, 1993, pp. 384–396.

[BHM+15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (May 2015), pp. 507–525.

[Bin97] D. Binkley. "Semantics Guided Regression Test Cost Reduction". In: *IEEE Transactions on Software Engineering* 23.8 (Aug. 1997), pp. 498–516.

[Bin98] D. Binkley. "The Application of Program Slicing to Regression Testing". In: *Information and Software Technology* 40.11–12 (1998), pp. 583–594.

[BKL+16] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr. "Reasoning About Product-Line Evolution Using Complex Feature Model Differences". In: *Automated Software Engineering* 23.4 (Dec. 2016), pp. 687–733.

[BKS11] D. Bruns, V. Klebanov, and I. Schaefer. "Verification of Software Product Lines with Delta-Oriented Slicing". In: *Formal Verification of Object-Oriented Software*. Ed. by B. Beckert and C. Marché. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 61–75.

[BL14] H. Baller and M. Lochau. "Towards Incremental Test Suite Optimization for Software Product Lines". In: *Proceedings of the 6th International Workshop on Feature-Oriented Software Development*. FOSD '14. New York, NY, USA: ACM, 2014, pp. 30–36.

[BLB+15] J. Bürdek, M. Lochau, S. Bauregger, A. Holzer, A. von Rhein, S. Apel, and D. Beyer. "Facilitating Reuse in Multi-Goal Test-Suite Generation for Software Product Lines". In: *Fundamental Approaches to Software Engineering*. Ed. by A. Egyed and I. Schaefer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 84–99.

[BLH09] L. Briand, Y. Labiche, and S. He. "Automating Regression Test Selection Based on UML Designs". In: *Information and Software Technology* 51.1 (2009). Special Section - Most Cited Articles in 2002 and Regular Research Papers, pp. 16–30.

[BLL+14] H. Baller, S. Lity, M. Lochau, and I. Schaefer. "Multi-Objective Test Suite Optimization for Incremental Product Family Testing". In: *International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2014, pp. 303–312.

[BLS02] L. C. Briand, Y. Labiche, and G. Soccar. "Automating Impact Analysis and Regression Test Selection Based on UML Designs". In: *International Conference on Software Maintenance*. Oct. 2002, pp. 252–261.

[BM16] H. Beohar and M. R. Mousavi. "Input–Output Conformance Testing for Software Product Lines". In: *Journal of Logical and Algebraic Methods in Programming* 85.6 (2016). NWPT 2013, pp. 1131–1153.

[BMS+11] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. "Regression Test Selection Techniques: A Survey". In: *Informatica* 35.3 (2011).

[Boh96] S. Bohner. "Impact Analysis in the Software Change Process: A Year 2000 Perspective". In: *Proceedings of the International Conference on Software Maintenance*. Nov. 1996, pp. 42–51.

[BP14] G. Botterweck and A. Pleuss. "Evolution of Software Product Lines". In: *Evolving Software Systems*. Springer, 2014, pp. 265–295.

[BPD+10] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, and S. Kowalewski. "EvoFM: Feature-driven Planning of Product-line Evolution". In: *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*. PLEASE '10. Cape Town, South Africa: ACM, 2010, pp. 24–31.

[BPP+09] G. Botterweck, A. Pleuss, A. Polzer, and S. Kowalewski. "Towards Feature-Driven Planning of Product-Line Evolution". In: *Proceedings of the International Workshop on Feature-Oriented Software Development*. FOSD '09. Denver, Colorado, USA: ACM, 2009, pp. 109–116.

[BSR04] D. Batory, J. N. Sarvela, and A. Rauschmayer. "Scaling Step-Wise Refinement". In: *IEEE Transactions on Software Engineering* 30.6 (June 2004), pp. 355–371.

[BSR10] D. Benavides, S. Segura, and A. Ruiz-Cortés. "Automated Analysis of Feature Models 20 Years Later: A Literature Review". In: *Information Systems* 35.6 (2010), pp. 615–636.

[BTG12] P. Borba, L. Teixeira, and R. Gheyi. "A Theory of Software Product Line Refinement". In: *Theoretical Computer Science* 455 (2012). International Colloquium on Theoretical Aspects of Computing 2010, pp. 2–30.

[BUK10] D. Batory, E. Uzuncaova, and S. Khurshid. "Incremental Test Generation for Software Product Lines". In: *IEEE Transactions on Software Engineering* 36.03 (May 2010), pp. 309–322.

[BW05] I. Brückner and H. Wehrheim. "Slicing an Integrated Formal Method for Verification". In: *Proceedings of the International Conference on Formal Methods and Software Engineering*. ICFEM'05. Manchester, UK: Springer-Verlag, 2005, pp. 360–374.

[CA05] K. Czarnecki and M. Antkiewicz. "Mapping Features to Models: A Template Approach Based on Superimposed Variants". In: *Generative Programming and Component Engineering*. Ed. by R. Glück and M. Lowry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 422–437.

[CAA09] L. Chen, M. Ali Babar, and N. Ali. "Variability Management in Software Product Lines: A Systematic Review". In: *Proceedings of the 13th International Software Product Line Conference*. SPLC '09. San Francisco, California, USA: Carnegie Mellon University, 2009, pp. 81–90.

[CAK+05] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, and K. Pietroszek. "Model-Driven Software Product Lines". In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 126–127.

[CCS+13] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking". In: *IEEE Transactions on Software Engineering* 39.8 (Aug. 2013), pp. 1069–1089.

[CD05] M. L. Crane and J. Dingel. "UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal". In: *Model Driven Engineering Languages and Systems*. Ed. by L. Briand and C. Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 97–112.

[CDD+16] C. Chesta, F. Damiani, L. Dobriakova, M. Guernieri, S. Martini, M. Nieke, V. Rodrigues, and S. Schuster. "A Toolchain for Delta-Oriented Modeling of Software Product Lines". In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by T. Margaria and B. Steffen. Cham: Springer International Publishing, 2016, pp. 497–511.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[CHS15] D. Clarke, M. Helvensteijn, and I. Schaefer. "Abstract Delta Modelling". In: *Mathematical Structures in Computer Science* 25.3 (2015), pp. 482–527.

[Cla10] A. Classen. *Modelling with FTS: A Collection of Illustrative Examples*. Tech. rep. P-CS-TR SPLMC-00000001. PReCISE Research Center, Univ. of Namur, 2010.

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[COL+11] H. Cichos, S. Oster, M. Lochau, and A. Schürr. "Model-Based Coverage-Driven Test Suite Generation for Software Product Lines". In: *Model Driven Engineering Languages and Systems*. Ed. by J. Whittle, T. Clark, and T. Kühne. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 425–439.

[Col06] R. Colgren. *Basic Matlab, Simulink And Stateflow*. AIAA (American Institute of Aeronautics & Ast, 2006.

[CPS02] Y. Chen, R. L. Probert, and D. P. Sims. "Specification-Based Regression Test Selection with Risk Analysis". In: *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '02. Toronto, Ontario, Canada: IBM Press, 2002, pp. 1–.

[CPU07a] Y. Chen, R. L. Probert, and H. Ural. "Model-based Regression Test Suite Generation Using Dependence Analysis". In: *Proceedings of the International Workshop on Advances in Model-based Testing*. A-MOST '07. London, United Kingdom: ACM, 2007, pp. 54–62.

[CPU07b] Y. Chen, R. L. Probert, and H. Ural. "Regression Test Suite Reduction Using Extended Dependence Analysis". In: *International Workshop on Software Quality Assurance*. SOQUA '07. Dubrovnik, Croatia: ACM, 2007, pp. 62–69.

[dCMC+14] I. do Carmo Machado, J. D. McGregor, Y. C. Cavalcanti, and E. S. de Almeida. "On Strategies for Testing Software Product Lines: A Systematic Literature Review". In: *Information and Software Technology* 56.10 (2014), pp. 1183–1199.

[DCP+12] X. Devroey, M. Cordy, G. Perrouin, E.-Y. Kang, P.-Y. Schobbens, P. Heymans, A. Legay, and B. Baudry. "A Vision for Behavioural Model-Driven Validation of Software Product Lines". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by T. Margaria and B. Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 208–222.

[DFG+17] F. Damiani, D. Faitelson, C. Gladisch, and S. Tyszberowicz. "A Novel Model-Based Testing Approach for Software Product Lines". In: *Software & Systems Modeling* 16.4 (Oct. 2017), pp. 1223–1251.

[DGR+10] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. "Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering". In: *J. Syst. Softw.* 83.7 (2010), pp. 1108–1122.

[DGT13] F. Damiani, C. Gladisch, and S. Tyszberowicz. "Refinement-Based Testing of Delta-Oriented Product Lines". In: *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '13. Stuttgart, Germany: ACM, 2013, pp. 135–140.

[Din17] N. Dintzner. "Feature-Oriented Evolution of Variant-Rich Software Systems". PhD thesis. TU Delft Software Engineering, 2017.

[DKvD+14] N. Dintzner, U. Kulesza, A. van Deursen, and M. Pinzger. "Evaluating Feature Change Impact on Multi-Product Line Configurations Using Partial Information". In: *Software Reuse for Dynamic Systems in the Cloud and Beyond*. Ed. by I. Schaefer and I. Stamelos. Cham: Springer International Publishing, 2014, pp. 1–16.

[DL16] F. Damiani and M. Lienhardt. "On Type Checking Delta-Oriented Product Lines". In: *Integrated Formal Methods*. Ed. by E. Ábrahám and M. Huisman. Cham: Springer International Publishing, 2016, pp. 47–62.

[dM S10] P. A. d. M. S. Neto. "A Regression Testing Approach for Software Product Lines Architectures". PhD thesis. Universidade Federal de Pernambuco, 2010.

[dMdCC+10] P. A. d. M. S. Neto, I. d. C. Machado, Y. C. Cavalcanti, E. S. d. Almeida, V. C. Garcia, and S. R. d. L. Meira. "A Regression Testing Approach for Software Product Lines Architectures". In: *Brazilian Symposium on Software Components, Architectures and Reuse*. Sept. 2010, pp. 41–50.

[dMdCC+12] P. A. d. M. S. Neto, I. d. C. Machado, Y. C. Cavalcanti, E. S. d. Almeida, V. C. Garcia, and S. R. d. L. Meira. "An Experimental Study to Evaluate a SPL Architecture Regression Testing Approach". In: *IEEE International Conference on Information Reuse Integration*. Aug. 2012, pp. 608–615.

[dMdCM+11] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. "A Systematic Mapping Study of Software Product Lines Testing". In: *Information and Software Technology* 53.5 (2011). Special Section on Best Papers from XP2010, pp. 407–423.

[DNG+08a] D. Dhungana, T. Neumayer, P. Gruenbacher, and R. Rabiser. "Supporting the Evolution of Product Line Architectures with Variability Model Fragments". In: *Working IEEE/IFIP Conference on Software Architecture*. Feb. 2008, pp. 327–330.

[DNG+08b] D. Dhungana, T. Neumayer, P. Grunbacher, and R. Rabiser. "Supporting Evolution in Model-Based Product Line Engineering". In: *International Software Product Line Conference*. Sept. 2008, pp. 319–328.

[dOdA15] R. P. d. Oliveira and E. S. d. Almeida. "Requirements Evolution in Software Product Lines: An Empirical Study". In: *Brazilian Symposium on Components, Architectures and Reuse Software*. Sept. 2015, pp. 1–10.

[DPC+13] X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, and P. Heymans. "Towards Statistical Prioritization for Software Product Lines Testing". In: *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: ACM, 2013, 10:1–10:7.

[DPC+14] X. Devroey, G. Perrouin, M. Cordy, M. Papadakis, A. Legay, and P.-Y. Schobbens. "A Variability Perspective of Mutation Analysis". In: *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 841–844.

[DPC+17] X. Devroey, G. Perrouin, M. Cordy, H. Samih, A. Legay, P.-Y. Schobbens, and P. Heymans. "Statistical Prioritization for Software Product Line Testing: An Experience Report". In: *Software & Systems Modeling* 16.1 (Feb. 2017), pp. 153–171.

[DPG+11] J. Díaz, J. Pérez, J. Garbajosa, and A. L. Wolf. "Change Impact Analysis in Product-Line Architectures". In: *Software Architecture*. Ed. by I. Crnkovic, V. Gruhn, and M. Book. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 114–129.

[DPL+14] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-Y. Schobbens, and P. Heymans. "Coverage Criteria for Behavioural Testing of Software Product Lines". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change (ISoLa)*. Ed. by T. Margaria and B. Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 336–350.

[DPL+15] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans. "Covering SPL Behaviour with Sampled Configurations: An Initial Assessment". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '15. Hildesheim, Germany: ACM, 2015, 59:59–59:66.

[DPL+16] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans. "Search-Based Similarity-Driven Behavioural SPL Testing". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 89–96.

[DPS14] X. Devroey, G. Perrouin, and P.-Y. Schobbens. "Abstract Test Case Generation for Behavioural Testing of Software Product Lines". In: *Proceedings of the International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2*. ACM. 2014, pp. 86–93.

[DS12] F. Damiani and I. Schaefer. "Family-Based Analysis of Type Safety for Delta-Oriented Software Product Lines". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by T. Margaria and B. Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 193–207.

[DSL+13] M. Dukaczewski, I. Schaefer, R. Lachmann, and M. Lochau. "Requirements-Based Delta-Oriented SPL Testing". In: *International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*. May 2013, pp. 49–52.

[DSV+07] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. "A Survey on Model-based Testing Approaches: A Systematic Review". In: *Proceedings of the ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*. WEASELTech '07. Atlanta, Georgia: ACM, 2007, pp. 31–36.

[DvDP17] N. Dintzner, A. van Deursen, and M. Pinzger. "Analysing the Linux Kernel Feature Model Changes Using FMDiff". In: *Software & Systems Modeling* 16.1 (Feb. 2017), pp. 55–76.

[DVG+17] J. P. Diniz, G. Vale, F. Gaia, and E. Figueiredo. "Evaluating Delta-Oriented Programming for Evolving Software Product Lines". In: *Proceedings of the International Workshop on Variability and Complexity in Software Design*. VACE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 27–33.

[EAB02] T. Elrad, O. Aldawud, and A. Bader. "Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design". In: *Generative Programming and Component Engineering*. Ed. by D. Batory, C. Consel, and W. Taha. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 189–201.

[EBA+11] A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. "Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models". In: *International Conference on Information Technology: New Generations*. Apr. 2011, pp. 291–298.

[Eng10a] E. Engström. "Regression Test Selection and Product Line System Testing". In: *International Conference on Software Testing, Verification and Validation*. Apr. 2010, pp. 512–515.

[Eng10b] E. Engström. "Exploring Regression Testing and Software Product Line Testing – Research and State of Practice". Licentiate Thesis. Lund University, 2010.

[ER11] E. Engström and P. Runeson. "Software Product Line Testing – A Systematic Mapping Study". In: *Information and Software Technology* 53.1 (2011), pp. 2–13.

[ERS10] E. Engström, P. Runeson, and M. Skoglund. "A Systematic Review on Regression Test Selection Techniques". In: *Information and Software Technology* 52.1 (2010), pp. 14–30.

[Esh09] R. Eshuis. "Reconciling Statechart Semantics". In: *Science of Computer Programming* 74.3 (2009), pp. 65–99.

[Far10] M. Farrag. "Colored Model Based Testing for Software Product Lines". PhD thesis. Technical University of Ilmenau, 2010.

[FBS+12] F. Ferreira, P. Borba, G. Soares, and R. Gheyi. "Making Software Product Line Evolution Safer". In: *Brazilian Symposium on Software Components, Architectures and Reuse*. Sept. 2012, pp. 21–30.

[FCS+08] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. "Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability". In: *Proceedings of the International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 261–270.

[FG08] A. Fantechi and S. Gnesi. "Formal Modeling for Product Families Engineering". In: *Proceedings of the International Software Product Line Conference*. SPLC'12. Sept. 2008, pp. 193–202.

[FGF+14] G. C. S. Ferreira, F. N. Gaia, E. Figueiredo, and M. de Almeida Maia. "On the Use of Feature-Oriented Programming for Evolving Software Product Lines — A Comparative Study". In: *Science of Computer Programming* 93 (2014). Special Issue with Selected Papers from the Brazilian Symposium on Programming Languages (SBLP 2011), pp. 65–85.

[FIM+07] Q.-a. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and A. Nadeem. "An Approach for Selective State Machine Based Regression Testing". In: *Proceedings of the International Workshop on Advances in Model-based Testing*. A-MOST '07. London, United Kingdom: ACM, 2007, pp. 44–52.

[FIM+10] Q.-a. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and M. Riebisch. "A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support". In: *Proceedings of the IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. ECBS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 41–49.

[FM06] T. Feng and J. I. Maletic. "Applying Dynamic Change Impact Analysis in Component-Based Architecture Design". In: *ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*. June 2006, pp. 43–48.

[FN08] M. Fahad and A. Nadeem. "A Survey of UML Based Regression Testing". In: *Intelligent Information Processing IV*. Ed. by Z. Shi, E. Mercier-Laurent, and D. Leake. Boston, MA: Springer US, 2008, pp. 200–210.

[FSM18] V. H. Fragal, A. Simao, and M. R. Mousavi. "Hierarchical Featured State Machines". In: *Science of Computer Programming* (2018).

[FTS13] W. Fenske, T. Thüm, and G. Saake. "A Taxonomy of Software Product Line Reengineering". In: *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: ACM, 2013, 4:1–4:8.

[FWT+11] M. Fisher II, J. Wloka, F. Tip, B. G. Ryder, and A. Luchansky. "An Evaluation of Change-Based Coverage Criteria". In: *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. PASTE '11. Szeged, Hungary: ACM, 2011, pp. 21–28.

[GB08] K. Gallagher and D. Binkley. "Program Slicing". In: *Frontiers of Software Maintenance*. Sept. 2008, pp. 58–67.

[GF11] N. Gamez and L. Fuentes. "Software Product Line Evolution with Cardinality-Based Feature Models". In: *Proceedings of the International Conference on Top Productivity Through Software Reuse*. ICSR'11. Pohang, South Korea: Springer-Verlag, 2011, pp. 102–118.

[GF13] N. Gamez and L. Fuentes. "Architectural Evolution of FamiWare Using Cardinality-Based Feature Models". In: *Inf. Softw. Technol.* 55.3 (Mar. 2013), pp. 563–580.

[GFF+14] F. N. Gaia, G. C. S. Ferreira, E. Figueiredo, and M. de Almeida Maia. "A Quantitative and Qualitative Assessment of Aspectual Feature Modules for Evolving Software Product Lines". In: *Science of Computer Programming* 96 (2014). Selected and extended papers of the Brazilian Symposium on Programming Languages 2012 (SBLP 2012), pp. 230–253.

[GG08] M. W. Godfrey and D. M. German. "The Past, Present, and Future of Software Evolution". In: *Frontiers of Software Maintenance*. Sept. 2008, pp. 129–138.

[GHP02] E. Gery, D. Harel, and E. Palachi. "Rhapsody: A Complete Life-Cycle Model-Based Development System". In: *Integrated Formal Methods*. Ed. by M. Butler, L. Petre, and K. Sere. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–10.

[GHS92] R. Gupta, M. J. Harrold, and M. L. Soffa. "An Approach to Regression Testing Using Slicing". In: *Proceedings of the International Conference on Software Maintenance*. Nov. 1992, pp. 299–308.

[GHS96] R. Gupta, M. J. Harrold, and M. L. Soffa. "Program Slicing-Based Regression Testing Techniques". In: *Software Testing, Verification and Reliability* 6.2 (1996), pp. 83–111.

[GL91] K. B. Gallagher and J. R. Lyle. "Using Program Slicing in Software Maintenance". In: *IEEE Transactions on Software Engineering* 17.8 (Aug. 1991), pp. 751–761.

[GLS08] A. Gruler, M. Leucker, and K. Scheidemann. "Modeling and Model Checking Software Product Lines". In: *Formal Methods for Open Object-Based Distributed Systems*. Ed. by G. Barthe and F. S. de Boer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 113–131.

[Gom06] H. Gomaa. "Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures". In: *Reuse of Off-the-Shelf Components*. Ed. by M. Morisio. Vol. 4039. Springer, 2006, pp. 440–440.

[GW10] J. Guo and Y. Wang. "Towards Consistent Evolution of Feature Models". In: *Proceedings of the International Conference on Software Product Lines*. SPLC'10. Jeju Island, South Korea: Springer-Verlag, 2010, pp. 451–455.

[GWT+12] J. Guo, Y. Wang, P. Trinidad, and D. Benavides. "Consistency Maintenance for Evolving Feature Models". In: *International Journal of Expert Systems with Applications* 39.5 (Apr. 2012), pp. 4987–4998.

[Har00] M. J. Harrold. "Testing: A Roadmap". In: *International Conference on Software Engineering*. Limerick, Ireland: ACM, 2000, pp. 61–72.

[Har87] D. Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274.

[HD95] M. Harman and S. Danicic. "Using Program Slicing to Simplify Testing". In: *Software Testing, Verification and Reliability* 5.3 (1995), pp. 143–162.

[HGB+18] I. Hajri, A. Goknil, L. C. Briand, and T. Stephany. "Change Impact Analysis for Evolving Configuration Decisions in Product Line Use Case Models". In: *Journal of Systems and Software* 139 (2018), pp. 211–237.

[HHD99] R. Hierons, M. Harman, and S. Danicic. "Using Program Slicing to Assist in the Detection of Equivalent Mutants". In: *Software Testing, Verification and Reliability* 9.4 (1999), pp. 233–262.

[HHF+02] R. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. "Conditioned Slicing Supports Partition Testing". In: *Software Testing, Verification and Reliability* 12.1 (2002), pp. 23–28.

[HHH+02] M. Harman, L. Hu, R. Hierons, C. Fox, S. Danicic, J. Wegener, H. Sthamer, and A. Baresel. "Evolutionary Testing Supported by Slicing and Transformation". In: *International Conference on Software Maintenance*. Oct. 2002, pp. 285–.

[HHK+13] A. Haber, K. Hölldobler, C. Kolassa, M. Look, B. Rumpe, K. Müller, and I. Schaefer. "Engineering Delta Modeling Languages". In: *Proceedings of the International Software Product Line Conference*. SPLC '13. Tokyo, Japan: ACM, 2013, pp. 22–31.

[HHK+15] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, I. Schaefer, and C. Schulze. "Systematic Synthesis of Delta Modeling Languages". In: *International Journal on Software Tools for Technology Transfer* 17.5 (Oct. 2015), pp. 601–626.

[HKM+13] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer. "First-Class Variability Modeling in Matlab/Simulink". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '13. Pisa, Italy: ACM, 2013, 4:1–4:8.

[HKR+11a] A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. "Delta-Oriented Architectural Variability Using MontiCore". In: *Proceedings of the European Conference on Software Architecture: Companion Volume*. ECSA '11. Essen, Germany: ACM, 2011, 6:1–6:10.

[HKR+11b] A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. "Delta-oriented Architectural Variability Using MontiCore". In: *Proceedings of the European Conference on Software Architecture*. ECSA '11. Essen, Germany: ACM, 2011, 6:1–6:10.

[HMO+08] O. Haugen, B. Moller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. "Adding Standardized Variability to Domain Specific Languages". In: *International Software Product Line Conference*. Sept. 2008, pp. 139–148.

[HN96] D. Harel and A. Naamad. "The STATEMATE Semantics of Statecharts". In: *ACM Trans. Softw. Eng. Methodol.* 5.4 (Oct. 1996), pp. 293–333.

[HO08] M. J. Harrold and A. Orso. "Retesting Software during Development and Maintenance". In: *Frontiers of Software Maintenance*. Sept. 2008, pp. 99–108.

[Hol12] H. Holdschick. "Challenges in the Evolution of Model-based Software Product Lines in the Automotive Domain". In: *Proceedings of the International Workshop on Feature-Oriented Software Development*. FOSD '12. Dresden, Germany: ACM, 2012, pp. 70–73.

[HP98] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. 1st. New York, NY, USA: McGraw-Hill, Inc., 1998.

[HPP+13] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. "Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing". In: *IEEE International Conference on Software Testing, Verification and Validation Workshops*. 2013, pp. 188–197.

[HPP+14] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. "Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines". In: *IEEE Transactions on Software Engineering* 40.7 (July 2014), pp. 650–670.

[HRG+12] W. Heider, R. Rabiser, P. Grünbacher, and D. Lettner. "Using Regression Testing to Analyze the Impact of Changes to Variability Models on Products". In: *Proceedings of the International Software Product Line Conference*. SPLC '12. Salvador, Brazil: ACM, 2012, pp. 196–205.

[HRG12] W. Heider, R. Rabiser, and P. Grünbacher. "Facilitating the Evolution of Products in Product Line Engineering by Capturing and Replaying Configuration Decisions". In: *International Journal on Software Tools for Technology Transfer* 14.5 (Oct. 2012), pp. 613–630.

[HRH+05] J. Hassine, J. Rilling, J. Hewitt, and R. Dssouli. "Change Impact Analysis for Requirement Evolution Using Use Case Maps". In: *International Workshop on Principles of Software Evolution*. Sept. 2005, pp. 81–90.

[HRR+12] A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. "Evolving Delta-Oriented Software Product Line Architectures". In: *Monterey Workshop*. Springer. 2012, pp. 183–208.

[HS88] M. J. Harrold and M. L. Souffa. "An Incremental Approach to Unit Testing during Maintenance". In: *Proceedings of the International Conference on Software Maintenance*. Oct. 1988, pp. 362–367.

[HWE17] M. Hamza, R. J. Walker, and M. Elaasar. "Unanticipated Evolution in Software Product Lines Versus Independent Products: A Case Study". In: *Proceedings of the International Systems and Software Product Line Conference*. SPLC '17. Sevilla, Spain: ACM, 2017, pp. 97–104.

[JG06] D. Jeffrey and N. Gupta. "Test Case Prioritization Using Relevant Slices". In: *Annual International Computer Software and Applications Conference*. Vol. 1. Sept. 2006, pp. 411–420.

[JG08] D. Jeffrey and N. Gupta. "Experiments with Test Case Prioritization Using Relevant Slices". In: *Journal of Systems and Software* 81.2 (2008). Model-Based Software Testing, pp. 196–221.

[JH11] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 649–678.

[JHF+12] M. F. Johansen, O. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. "Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines". In: *Model Driven Engineering Languages and Systems*. Ed. by R. B. France, J. Kazmeier, R. Breu, and C. Atkinson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 269–284.

[JHF12] M. F. Johansen, O. Haugen, and F. Fleurey. "An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models". In: *Proceedings of the International Software Product Line Conference - Volume 1*. SPLC '12. Salvador, Brazil: ACM, 2012, pp. 46–55.

[JM13] N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Springer Science & Business Media, 2013.

[JWZ02] W. Ji, D. Wei, and Q. Zhi-Chang. "Slicing Hierarchical Automata for Model Checking UML Statecharts". In: *Formal Methods and Software Engineering*. Ed. by C. George and H. Miao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 435–446.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Carnegie-Mellon University - Software Engineering Institute, 1990.

[KHS+14] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. "DeltaJ 1.5: Delta-oriented Programming for Java 1.5". In: *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. Cracow, Poland: ACM, 2014, pp. 63–74.

[KJM+17] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani. "Effective Regression Test Case Selection: A Systematic Literature Review". In: *ACM Computing Survey* 50.2 (May 2017), 29:1–29:32.

[KLB12] J. Kamischke, M. Lochau, and H. Baller. "Conditioned Model Slicing of Feature-Annotated State Machines". In: *Proceedings of the International Workshop on Feature-Oriented Software Development*. FOSD '12. Dresden, Germany: ACM, 2012, pp. 9–16.

[KLL+14] M. Kowal, C. Legat, D. Lorefice, C. Prehofer, I. Schaefer, and B. Vogel-Heuser. "Delta Modeling for Variant-Rich and Evolving Manufacturing Systems". In: *Proceedings of the International Workshop on Modern Software Engineering Methods for Industrial Automation*. MoSEMInA 2014. Hyderabad, India: ACM, 2014, pp. 32–41.

[KMS+83] J. Kramer, J. Magee, M. Sloman, and A. Lister. "CONIC: An Integrated Approach to Distributed Computer Control Systems". In: *IEEE Computers and Digital Techniques* 130.1 (Jan. 1983), pp. 1–.

[Kru02] C. Krueger. "Eliminating the Adoption Barrier". In: *IEEE Software* 19.4 (July 2002), pp. 29–31.

[KS14] F. Kanning and S. Schulze. "Program Slicing in the Presence of Preprocessor Variability". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. Sept. 2014, pp. 501–505.

[KST+16] S. Krieter, R. Schröter, T. Thüm, W. Fenske, and G. Saake. "Comparing Algorithms for Efficient Feature-Model Slicing". In: *Proceedings of the 20th International Systems and Software Product Line Conference*. SPLC '16. Beijing, China: ACM, 2016, pp. 60–64.

[KT04] B. Korel and L. H. Tahat. "Understanding Modifications in State-Based Models". In: *Proceedings. of the IEEE International Workshop on Program Comprehension*. June 2004, pp. 246–250.

[KTT+15] M. Kowal, M. Tschaikowski, M. Tribastone, and I. Schaefer. "Scaling Size and Parameter Spaces in Variability-Aware Software Performance Models (T)". In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2015, pp. 407–417.

[KTV02] B. Korel, L. H. Tahat, and B. Vaysburg. "Model-Based Regression Test Reduction Using Dependence Analysis". In: *Proceedings of the International Conference on Software Maintenance*. Oct. 2002, pp. 214–223.

[Lac17] R. Lachmann. "Black-Box Test Case Selection and Prioritization for Software Variants and Versions". PhD thesis. Technische Universität Braunschweig, Nov. 2017.

[LAT+17] S. Lity, M. Al-Hajjaji, T. Thüm, and I. Schaefer. "Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-line Analysis". In: *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. VaMoS '17. Eindhoven, Netherlands: ACM, 2017, pp. 60–67.

[LBG17] L. Linsbauer, T. Berger, and P. Grünbacher. "A Classification of Variation Control Systems". In: *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2017. Vancouver, BC, Canada: ACM, 2017, pp. 49–62.

[LBL+14] M. Lochau, J. Bürdek, S. Lity, M. Hagner, C. Legat, U. Goltz, and A. Schürr. "Applying Model-Based Software Product Line Testing Approaches to the Automation Engineering Domain". In: *at-Automatisierungstechnik* 62.11 (2014), pp. 771–780.

[LBL+15] S. Lity, J. Bürdek, M. Lochau, M. Berens, A. Schürr, and I. Schaefer. "Re-Engineering Automation Systems as Dynamic Software Product Lines". In: *Dagstuhl-Workshop Model-Based Development of Embedded Systems (MBEES)*. Ed. by M. Riebisch, M. Huhn, J. Philipps, and B. Schätz. Vol. 11. 2015.

[LBL+17] R. Lachmann, S. Beddig, S. Lity, S. Schulze, and I. Schaefer. "Risk-Based Integration Testing of Software Product Lines". In: *International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. VaMoS '17. Eindhoven, Netherlands: ACM, 2017, pp. 52–59.

[LBS15] S. Lity, H. Baller, and I. Schaefer. "Towards Incremental Model Slicing for Delta-Oriented Software Product Lines". In: *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. ERA Track. Mar. 2015, pp. 530–534.

[LC13] M. A. Laguna and Y. Crespo. "A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring". In: *Science of Computer Programming* 78.8 (2013). Special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages, pp. 1010–1034.

[LDT+18] M. Lienhardt, F. Damiani, L. Testa, and G. Turin. "On Checking Delta-Oriented Product Lines of Statecharts". In: *Science of Computer Programming* 166 (2018), pp. 3–34.

[Leh11a] S. Lehnert. *A Review of Software Change Impact Analysis*. Tech. rep. Ilmenau University of Technology, 2011.

[Leh11b] S. Lehnert. "A Taxonomy for Software Change Impact Analysis". In: *Proceedings of the International Workshop on Principles of Software Evolution and the Annual ERCIM Workshop on Software Evolution*. IWPSE-EVOL '11. Szeged, Hungary: ACM, 2011, pp. 41–50.

[Leh96] M. M. Lehman. "Laws of Software Evolution Revisited". In: *Software Process Technology*. Ed. by C. Montangero. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 108–124.

[LFC+16] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. "Evolutionary Computation for Software Product Line Testing: An Overview and Open Challenges". In: *Computational Intelligence and Quantitative Software Engineering*. Ed. by W. Pedrycz, G. Succi, and A. Sillitti. Cham: Springer International Publishing, 2016, pp. 59–87.

[LFH+15] J. Ladiges, A. Fay, C. Haubeck, W. Lamersdorf, S. Lity, and I. Schaefer. "Supporting Commissioning of Production Plants by Model-Based Testing and Model Learning". In: *International Symposium on Industrial Electronics (ISIE)*. June 2015, pp. 606–611.

[LFM96] A. D. Lucia, A. R. Fasolino, and M. Munro. "Understanding Function Behaviors through Program Slicing". In: *Workshop on Program Comprehension*. Mar. 1996, pp. 9–18.

[LFR+15] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed. "A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines". In: *IEEE International Conference on Software Testing, Verification and Validation Workshops*. Apr. 2015, pp. 1–10.

[LG10] M. Lochau and U. Goltz. "Feature Interaction Aware Test Case Generation for Embedded Control Systems". In: *Electronic Notes in Theoretical Computer Science* 264.3 (2010). Proceedings of the Sixth Workshop on Model-Based Testing (MBT 2010), pp. 37–52.

[LGC+15] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman. "SAT-Based Analysis of Large Real-World Feature Models Is Easy". In: *Proceedings of the International Conference on Software Product Lines*. SPLC '15. Nashville, Tennessee: ACM, 2015, pp. 91–100.

[LGS+19] L. Luthmann, T. Gerecht, A. Stephan, J. Bürdek, and M. Lochau. "Minimum/Maximum Delay Testing of Product Lines with Unbounded Parametric Real-Time Constraints". In: *Journal of Systems and Software* 149 (2019), pp. 535–553.

[LJC+14] R. E. Lopez-Herrejon, J. Javier Ferrer, F. Chicano, E. N. Haslinger, A. Egyed, and E. Alba. "A Parallel Evolutionary Algorithm for Prioritized Pairwise Testing of Software Product Lines". In: *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*. GECCO '14. Vancouver, BC, Canada: ACM, 2014, pp. 1255–1262.

[LK12] M. Lochau and J. Kamischke. "Parameterized Preorder Relations for Model-Based Testing of Software Product Lines". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by T. Margaria and B. Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 223–237.

[LKL12] J. Lee, S. Kang, and D. Lee. "A Survey on Software Product Line Testing". In: *Proceedings of the International Software Product Line Conference*. SPLC '12. Salvador, Brazil: ACM, 2012, pp. 31–40.

[LKS16] S. Lity, M. Kowal, and I. Schaefer. "Higher-Order Delta Modeling for Software Product Line Evolution". In: *International Workshop on Feature-Oriented Software Development*. FOSD 2016 (FOSD@SPLASH). Amsterdam, Netherlands: ACM, 2016, pp. 39–48.

[LLA+16] R. Lachmann, S. Lity, M. Al-Hajjaji, F. Fürchtegott, and I. Schaefer. "Fine-Grained Test Case Prioritization for Integration Testing of Delta-Oriented Software Product Lines". In: *International Workshop on Feature-Oriented Software Development*. FOSD 2016 (FOSD@SPLASH). Amsterdam, Netherlands: ACM, 2016, pp. 1–10.

[LLL+14] M. Lochau, S. Lity, R. Lachmann, I. Schaefer, and U. Goltz. "Delta-Oriented Model-Based Integration Testing of Large-Scale Systems". In: *Journal of Systems and Software* 91 (2014), pp. 63–84.

[LLL+15] R. Lachmann, S. Lity, S. Lischke, S. Beddig, S. Schulze, and I. Schaefer. "Delta-Oriented Test Case Prioritization for Integration Testing of Software Product Lines". In: *International Software Product Line Conference (SPLC)*. SPLC '15. Nashville, Tennessee: ACM, 2015, pp. 81–90.

[LLS+12] S. Lity, M. Lochau, I. Schaefer, and U. Goltz. "Delta-Oriented Model-Based SPL Regression Testing". In: *International Workshop on Product LinE Approaches in Software Engineering (PLEASE@ICSE)*. June 2012, pp. 53–56.

[LM08] J. T. Lalchandani and R. Mall. "Regression Testing Based-on Slicing of Component-based Software Architectures". In: *Proceedings of the India Software Engineering Conference*. ISEC '08. Hyderabad, India: ACM, 2008, pp. 67–76.

[LMB+14] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. "DeltaCCS: A Core Calculus for Behavioral Change". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by T. Margaria and B. Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 320–335.

[LMB+16] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. "Incremental Model Checking of Delta-Oriented Software Product Lines". In: *Journal of Logical and Algebraic Methods in Programming* 85.1, Part 2 (2016). Formal Methods for Software Product Line Engineering, pp. 245–267.

[LML15] L. Luthmann, S. Mennicke, and M. Lochau. "Towards an I/O Conformance Testing Theory for Software Product Lines based on Modal Interface Automata". In: *Proceedings Workshop on Formal Methods and Analysis in SPL Engineering*. 2015.

[LMT+16] S. Lity, T. Morbach, T. Thüm, and I. Schaefer. "Applying Incremental Model Slicing to Product-Line Regression Testing". In: *ICSR 2016: Software Reuse: Bridging with Social-Awareness (ICSR)*. Ed. by G. M. Kapitsaki and E. Santana de Almeida. Cham: Springer International Publishing, 2016, pp. 3–19.

[LNT+18] S. Lity, S. Nahrendorf, T. Thüm, C. Seidl, and I. Schaefer. "175% Modeling for Product-Line Evolution of Domain Artifacts". In: *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. VaMoS 2018. Madrid, Spain: ACM, 2018, pp. 27–34.

[LNT+19] S. Lity, M. Nieke, T. Thüm, and I. Schaefer. "Retest Test Selection for Product-Line Regression Testing of Variants and Versions of Variants". In: *Journal of Systems and Software* 147 (2019), pp. 46–63.

[Loc13] M. Lochau. "Model-Based Conformance Testing of Software Product Lines". PhD thesis. University of Braunschweig - Institute of Technology, 2013.

[LOG+12] M. Lochau, S. Oster, U. Goltz, and A. Schürr. "Model-Based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering". In: *Software Quality Journal* 20.3 (Sept. 2012), pp. 567–604.

[LPK+14] M. Lochau, S. Peldszus, M. Kowal, and I. Schaefer. "Model-Based Testing". In: *Advanced Lectures of the 14th International School on Formal Methods for Executable Software Models - Volume 8483*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 310–342.

[LRB+19] M. Lochau, D. Reuling, J. Bürdek, T. Kehrer, S. Lity, A. Schürr, and U. Kelter. "Model-Based Round-Trip Engineering and Testing of Evolving Software Product Lines". In: *Managed Software Evolution*. Ed. by R. Reussner, M. Goedicke, W. Hasselbring, B. Vogel-Heuser, J. Keim, and L. Märtin. Cham: Springer International Publishing, 2019, pp. 141–173.

[LS14] H. Lackner and M. Schmidt. "Towards the Assessment of Software Product Line Tests: A Mutation System for Variable Systems". In: *Proceedings of the International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*. SPLC '14. ACM, 2014, pp. 62–69.

[LSB+17] L. Luthmann, A. Stephan, J. Bürdek, and M. Lochau. "Modeling and Testing Product Lines with Unbounded Parametric Real-Time Constraints". In: *Proceedings of the International Systems and Software Product Line Conference - Volume A*. SPLC '17. Sevilla, Spain: ACM, 2017, pp. 104–113.

[LSK+12] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. "Incremental Model-Based Testing of Delta-Oriented Software Product Lines". In: *Tests and Proofs (TAP)*. Ed. by A. D. Brucker and J. Julliand. Vol. 7305. LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 67–82.

[LSK+13] G. Lima, J. Santos, U. Kulesza, D. Alencar, and S. V. Fialho. "A Delta Oriented Approach to the Evolution and Reconciliation of Enterprise Software Products Lines". In: *Proceedings of the International Conference on Enterprise Information Systems*. INSTICC. SciTePress, 2013, pp. 255–263.

[LSL+13] B. Li, X. Sun, H. Leung, and S. Zhang. "A Survey of Code-Based Change Impact Analysis Techniques". In: *Software Testing, Verification and Reliability* 23.8 (2013), pp. 613–646.

[LSR07] F. J. v. d. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Berlin, Heidelberg: Springer-Verlag, 2007.

[LTW+14] H. Lackner, M. Thomas, F. Wartenberg, and S. Weißleder. "Model-Based Test Design of Product Lines: Raising Test Design to the Product Line Level". In: *International Conference on Software Testing, Verification and Validation*. Mar. 2014, pp. 51–60.

[Luc01] A. D. Lucia. "Program Slicing: Methods and Applications". In: *Proceedings IEEE International Workshop on Source Code Analysis and Manipulation*. Nov. 2001, pp. 142–149.

[LvRK+13] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. "Scalable Analysis of Variable Software". In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 81–91.

[LW89] H. K. N. Leung and L. White. "Insights into Regression Testing (Software Testing)". In: *Proceedings of the International Conference on Software Maintenance*. Oct. 1989, pp. 60–69.

[LW91] H. K. N. Leung and L. White. "A Cost Model to Compare Regression Test Strategies". In: *Proceedings of the International Conference on Software Maintenance*. Oct. 1991, pp. 201–208.

[Man02] M. Mannion. "Using First-Order Logic for Product Line Model Validation". In: *International Conference on Software Product Lines*. Ed. by G. J. Chastek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 176–187.

[MBB16] J. Maâzoun, N. Bouassida, and H. Ben-Abdallah. "Change Impact Analysis for Software Product Lines". In: *Journal of King Saud University - Computer and Information Sciences* 28.4 (2016), pp. 364–380.

[McG10] J. D. McGregor. "Testing a Software Product Line". In: *Testing Techniques in Software Engineering: Second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007, Revised Lectures*. Ed. by P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcook. Berlin, Heidelberg: Springer, 2010, pp. 104–140.

[MD16] L. Montalvillo and O. Diaz. "Requirement-Driven Evolution in Software Product Lines: A Systematic Mapping Study". In: *Journal of Systems and Software* 122 (2016), pp. 110–143.

[MI07] J. D. McGregor and K. Im. "The Implications of Variation for Testing in a Software Product Line". In: *International Conference of Software Product Lines*. 2007, pp. 59–64.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall international series in computer science. Prentice Hall, 1989.

[MNS+18] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. "Context-Aware Reconfiguration in Evolving Software Product Lines". In: *Science of Computer Programming* 163 (2018), pp. 139–159.

[MPC16] R. Muschevici, J. Proença, and D. Clarke. "Feature Nets: Behavioural Modelling of Software Product Lines". In: *Software & Systems Modeling* 15.4 (Oct. 2016), pp. 1181–1206.

[MSC14] T. Mens, A. Serebrenik, and A. Cleve, eds. *Evolving Software Systems*. Springer, 2014.

[MTN11] N. Mansour, H. Takkoush, and A. Nehme. "UML-Based Regression Testing for OO Software". In: *Journal of Software Maintenance and Evolution: Research and Practice* 23.1 (2011), pp. 51–68.

[MTS+17] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017.

[Muc07] H. Muccini. "Using Model Differencing for Architecture-Level Regression Testing". In: *EUROMICRO Conference on Software Engineering and Advanced Applications*. Aug. 2007, pp. 59–66.

[MWC09] M. Mendonca, A. Wąsowski, and K. Czarnecki. "SAT-based Analysis of Feature Models is Easy". In: *Proceedings of the International Software Product Line Conference*. SPLC '09. San Francisco, California, USA: Carnegie Mellon University, 2009, pp. 231–240.

[MWK+16] J. Meinicke, C. Wong, C. Kästner, T. Thüm, and G. Saake. "On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems". In: *IEEE/ACM International Conference on Automated Software Engineering*. Sept. 2016, pp. 483–494.

[NBA+15] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. "Safe Evolution Templates for Software Product Lines". In: *Journal of Systems and Software* 106 (2015), pp. 42–58.

[NES17] M. Nieke, G. Engel, and C. Seidl. "DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*. VAMOS '17. Eindhoven, Netherlands: ACM, 2017, pp. 92–99.

[NLS18] S. Nahrendorf, S. Lity, and I. Schaefer. *Applying Higher-Order Delta Modeling for the Evolution of Delta-Oriented Software Product Lines*. Tech. rep. 2018-01. TU Braunschweig, 2018. URL: https://www.isf.cs.tu-bs.de/cms/team/lity/TUBS_Report_2018-01_Nahrendorf_et_al.pdf.

[NMS+18] M. Nieke, J. Mauro, C. Seidl, T. Thüm, I. C. Yu, and F. Franzke. "Anomaly Analyses for Feature-Model Evolution". In: *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: ACM, 2018, pp. 188–201.

[NSS16] M. Nieke, C. Seidl, and S. Schuster. "Guaranteeing Configuration Validity in Evolving Software Product Lines". In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 73–80.

[NST18] M. Nieke, C. Seidl, and T. Thüm. "Back to the Future: Avoiding Paradoxes in Feature-Model Evolution". In: *Proceedings of the International Systems and Software Product Line Conference*. SPLC '18. Gothenburg, Sweden: ACM, 2018, pp. 48–51.

[NTS+11] L. Neves, L. Teixeira, D. Sena, V. Alves, U. Kulesza, and P. Borba. "Investigating the Safe Evolution of Software Product Lines". In: *Proceedings of the ACM International Conference on Generative Programming and Component Engineering*. GPCE '11. Portland, Oregon, USA: ACM, 2011, pp. 33–42.

[NZR09] L. Naslavsky, H. Ziv, and D. J. Richardson. "A Model-Based Regression Test Selection Technique". In: *IEEE International Conference on Software Maintenance*. Sept. 2009, pp. 515–518.

[NZR10] L. Naslavsky, H. Ziv, and D. J. Richardson. "MbSRT2: Model-Based Selective Regression Testing with Traceability". In: *International Conference on Software Testing, Verification and Validation*. Apr. 2010, pp. 89–98.

[OAH03] A. Orso, T. Apiwattanapong, and M. J. Harrold. "Leveraging Field Data for Impact Analysis and Regression Testing". In: *SIGSOFT Software Engineering Notes* 28.5 (Sept. 2003), pp. 128–137.

[Obj09] Object Management Group (OMG). *Unified Modeling Language (UML) Specification, Version 2.2*. OMG Document Number formal/09-02-02 (`https://www.omg.org/spec/UML/2.2/`). Jan. 2009.

[Off11] J. Offutt. "A Mutation Carol: Past, Present and Future". In: *Information and Software Technology* 53.10 (2011). Special Section on Mutation Testing, pp. 1098–1107.

[Oli08] E. M. Olimpiew. "Model-Based Testing for Software Product Lines". PhD thesis. George Mason University, 2008.

[OMR10] S. Oster, F. Markert, and P. Ritter. "Automated Incremental Pairwise Testing of Software Product Lines". In: *Software Product Lines: Going Beyond*. Ed. by J. Bosch and J. Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 196–210.

[OSH01] A. Orso, S. Sinha, and M. J. Harrold. "Incremental Slicing Based on Data-Dependences Types". In: *Proceedings of the IEEE International Conference on Software Maintenance*. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 158–.

[OT93] L. M. Ott and J. J. Thuss. "Slice Based Metrics for Estimating Cohesion". In: *Proceedings International Software Metrics Symposium*. May 1993, pp. 71–81.

[OWE+11] S. Oster, A. Wübbeke, G. Engels, and A. Schürr. "A Survey of Model-Based Software Product Lines Testing". In: *Model-based Testing for Embedded Systems*. CRC Press, 2011, pp. 338–381.

[OZL+11] S. Oster, M. Zink, M. Lochau, and M. Grechanik. "Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations". In: *Proceedings of the International Software Product Line Conference*. SPLC '11. Munich, Germany: ACM, 2011, 6:1–6:8.

[OZM+11] S. Oster, I. Zorcic, F. Markert, and M. Lochau. "MoSo-PoLiTe: Tool Support for Pairwise and Model-Based Software Product Line Testing". In: *Proceedings of the Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '11. Namur, Belgium: ACM, 2011, pp. 79–82.

[PBD+12] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski. "Model-Driven Support for Product Line Evolution on Feature Level". In: *Journal of Systems and Software* 85.10 (2012). Automated Software Evolution, pp. 2261–2274.

[PBvdL05] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media, 2005.

[PD07] R. Pilitowski and A. Dereziñska. "Code Generation and Execution Framework for UML 2.0 Classes and State Machines". In: *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*. Ed. by T. Sobh. Dordrecht: Springer Netherlands, 2007, pp. 421–427.

[PDŠ12] P. Paskevicius, R. Damasevicius, and V. Štuikys. "Change Impact Analysis of Feature Models". In: *Information and Software Technologies*. Ed. by T. Skersys, R. Butleris, and R. Butkiene. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 108–122.

[PKK+15] C. Pietsch, T. Kehrer, U. Kelter, D. Reuling, and M. Ohrndorf. "SiPL – A Delta-Based Modeling Framework for Software Product Line Engineering". In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2015, pp. 852–857.

[PM10] C. R. Panigrahi and R. Mall. "Model-Based Regression Test Case Prioritization". In: *SIGSOFT Software Engineering Notes* 35.6 (Nov. 2010), pp. 1–7.

[POK+17] C. Pietsch, M. Ohrndorf, U. Kelter, and T. Kehrer. "Incrementally Slicing Editable Submodels". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 913–918.

[POS+12] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon. "Pairwise Testing for Software Product Lines: Comparison of Two Approaches". In: *Software Quality Journal* 20.3 (Sept. 2012), pp. 605–643.

[PRK+17] C. Pietsch, D. Reuling, U. Kelter, and T. Kehrer. "A Tool Environment for Quality Assurance of Delta-Oriented Model-Based SPLs". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*. VAMOS '17. Eindhoven, Netherlands: ACM, 2017, pp. 84–91.

[PSK+10] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. l. Traon. "Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines". In: *International Conference on Software Testing, Verification and Validation*. Apr. 2010, pp. 459–468.

[PSM12] A. Petrenko, A. Simao, and J. C. Maldonado. "Model-Based Testing of Software and Systems: Recent Advances and Challenges". In: *International Journal on Software Tools for Technology Transfer* 14.4 (Aug. 2012), pp. 383–386.

[PSS+16] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed. "Multi-Objective Test Case Prioritization in Highly Configurable Systems: A Case Study". In: *Journal of Systems and Software* 122 (2016), pp. 287–310.

[PYZ11] X. Peng, Y. Yu, and W. Zhao. "Analyzing Evolution of Variability in a Software Product Line: From Contexts and Requirements to Features". In: *Information and Software Technology* 53.7 (2011), pp. 707–721.

[QCR08] X. Qu, M. B. Cohen, and G. Rothermel. "Configuration-Aware Regression Testing: An Empirical Study of Sampling and Prioritization". In: *Proceedings of the International Symposium on Software Testing and Analysis*. ISSTA '08. Seattle, WA, USA: ACM, 2008, pp. 75–86.

[QCW07] X. Qu, M. B. Cohen, and K. M. Woolf. "Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization". In: *IEEE International Conference on Software Maintenance*. Oct. 2007, pp. 255–264.

[QPB+14] C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, and G. Botterweck. "Consistency Checking for the Evolution of Cardinality-Based Feature Models". In: *Proceedings of the International Software Product Line Conference*. SPLC '14. Florence, Italy: ACM, 2014, pp. 122–131.

[RBR+15] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, and U. Kelter. "Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation". In: *Proceedings of the International Conference on Software Product Line*. SPLC '15. Nashville, Tennessee: ACM, 2015, pp. 131–140.

[RE12a] P. Runeson and E. Engström. "Software Product Line Testing – A 3D Regression Testing Problem". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Apr. 2012, pp. 742–746.

[RE12b] P. Runeson and E. Engström. "Chapter 7 - Regression Testing in Software Product Line Engineering". In: ed. by A. Hurson and A. Memon. Vol. 86. Advances in Computers. Elsevier, 2012, pp. 223–263.

[Rei94] G. Reinelt. *The Traveling Salesman - Computational Solutions for TSP Applications*. Vol. 840. LNCS. Springer, 1994.

[RH94] G. Rothermel and M. J. Harrold. "Selecting Tests and Identifying Test Coverage Requirements for Modified Software". In: *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '94. Seattle, Washington, USA: ACM, 1994, pp. 169–184.

[RH96] G. Rothermel and M. J. Harrold. "Analyzing Regression Test Selection Techniques". In: *IEEE Transactions on Software Engineering* 22.8 (Aug. 1996), pp. 529–551.

[RKP+05] A. Reuys, E. Kamsties, K. Pohl, and S. Reis. "Model-Based System Testing of Software Product Families ". In: *Advanced Information Systems Engineering*. Ed. by O. Pastor and J. Falcão e Cunha. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 519–534.

[RLB+18] S. Ruland, L. Luthmann, J. Bürdek, S. Lity, T. Thüm, M. Lochau, and M. Ribeiro. "Measuring Effectiveness of Sample-Based Product-Line Testing". In: *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: ACM, 2018, pp. 119–133.

[RMP07] S. Reis, A. Metzger, and K. Pohl. "Integration Testing in Software Product Line Engineering: A Model-Based Technique". In: *Fundamental Approaches to Software Engineering*. Ed. by M. B. Dwyer and A. Lopes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 321–335.

[SB99] M. Svahnberg and J. Bosch. "Evolution in Software Product Lines: Two Cases". In: *Journal of Software Maintenance* 11.6 (1999), pp. 391–422.

[SBB+10] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Proceedings of the International Software Product Line Conference*. Ed. by J. Bosch and J. Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 77–91.

[SBT16] G. Sampaio, P. Borba, and L. Teixeira. "Partially Safe Evolution of Software Product Lines". In: *Proceedings of the International Systems and Software Product Line Conference*. SPLC '16. Beijing, China: ACM, 2016, pp. 124–133.

[Sch04] K. Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[Sch10] I. Schaefer. "Variability Modelling for Model-Driven Development of Software Product Lines". In: *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*. 2010, pp. 85–92.

[SD07] M. Sinnema and S. Deelstra. "Classifying Variability Modeling Techniques". In: *Information and Software Technology* 49.7 (2007), pp. 717–739.

[SD10] I. Schaefer and F. Damiani. "Pure Delta-Oriented Programming". In: *Proceedings of the International Workshop on Feature-Oriented Software Development*. FOSD '10. Eindhoven, The Netherlands: ACM, 2010, pp. 49–56.

[Sei17] C. Seidl. "Integrated Management of Variability in Space and Time in Software Families". PhD thesis. Dresden University of Technology, Germany, 2017.

[SHA12] C. Seidl, F. Heidenreich, and U. Aßmann. "Co-Evolution of Models and Feature Mapping in Software Product Lines". In: *Proceedings of the International Software Product Line Conference*. SPLC '12. Salvador, Brazil: ACM, 2012, pp. 76–85.

[SHT06] P. Schobbens, P. Heymans, and J. Trigaux. "Feature Diagrams: A Survey and a Formal Semantics". In: *14th IEEE International Requirements Engineering Conference (RE'06)*. Sept. 2006, pp. 139–148.

[Sil12] J. Silva. "A Vocabulary of Program Slicing-Based Techniques". In: *ACM Computing Survey* 44.3 (June 2012), 12:1–12:41.

[SJ04] K. Schmid and I. John. "A Customizable Approach to Full Lifecycle Variability Management". In: *Science of Computer Programming* 53.3 (2004). Software Variability Management, pp. 259–284.

[SK13] M. Shirole and R. Kumar. "UML Behavioral Model Based Test Case Generation: A Survey". In: *SIGSOFT Software Engineering Notes* 38.4 (July 2013), pp. 1–13.

[SK14] H. Sabouri and R. Khosravi. "Reducing the Verification Cost of Evolving Product Families Using Static Analysis Techniques". In: *Science of Computer Programming* 83 (2014). Formal Aspects of Component Software (FACS 2011 selected & extended papers), pp. 35–55.

[SKT+16] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake. "Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. May 2016, pp. 667–678.

[SLS11] A. Spillner, T. Linz, and H. Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. 3rd. Rocky Nook, 2011.

[SOM10] A. Schürr, S. Oster, and F. Markert. "Model-Driven Software Product Line Testing: An Integrated Approach". In: *SOFSEM: Theory and Practice of Computer Science*. Ed. by J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 112–131.

[Som10] I. Sommerville. *Software Engineering*. 9th. Addison-Wesley Publishing Company, 2010.

[SPB+12] M. Schubanz, A. Pleuss, G. Botterweck, and C. Lewerentz. "Modeling Rationale over Time to Support Product Line Evolution Planning". In: *Proceedings of the International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '12. Leipzig, Germany: ACM, 2012, pp. 193–199.

[SPP+13] M. Schubanz, A. Pleuss, L. Pradhan, G. Botterweck, and A. K. Thurimella. "Model-Driven Planning and Monitoring of Long-Term Software Product Line Evolution". In: *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '13. Pisa, Italy: ACM, 2013, 18:1–18:5.

[SRC+12] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. "Software Diversity: State of the Art and Perspectives". In: *International Journal on Software Tools for Technology Transfer* 14.5 (Oct. 2012), pp. 477–495.

[SRG11] K. Schmid, R. Rabiser, and P. Grünbacher. "A Comparison of Decision Modeling Approaches in Product Lines". In: *Proceedings of the Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS '11. Namur, Belgium: ACM, 2011, pp. 119–126.

[SRS13] S. Schulze, O. Richers, and I. Schaefer. "Refactoring Delta-Oriented Software Product Lines". In: *Proceedings of the Annual International Conference on Aspect-oriented Software Development*. AOSD '13. Fukuoka, Japan: ACM, 2013, pp. 73–84.

[SSA13a] C. Seidl, I. Schaefer, and U. Aßmann. "Capturing Variability in Space and Time with Hyper Feature Models". In: *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: ACM, 2013, 6:1–6:8.

[SSA13b] C. Seidl, I. Schaefer, and U. Aßmann. "Variability-Aware Safety Analysis Using Delta Component Fault Diagrams". In: *Proceedings of the International Software Product Line Conference Co-Located Workshops*. SPLC '13 Workshops. Tokyo, Japan: ACM, 2013, pp. 2–9.

[SSA14a] C. Seidl, I. Schaefer, and U. Aßmann. "DeltaEcore-A Model-Based Delta Language Generation Framework". In: *Modellierung 2014*. Ed. by H.-G. Fill, D. Karagiannis, and U. Reimer. Bonn: Gesellschaft für Informatik e.V., 2014, pp. 81–96.

[SSA14b] C. Seidl, I. Schaefer, and U. Aßmann. "Integrated Management of Variability in Space and Time in Software Families". In: *Proceedings of the International Software Product Line Conference*. SPLC '14. Florence, Italy: ACM, 2014, pp. 22–31.

[SSR14] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. "A Comparison of Test Case Prioritization Criteria for Software Product Lines". In: *IEEE International Conference on Software Testing, Verification and Validation*. Mar. 2014, pp. 41–50.

[ST00] S. R. Schach and A. Tomer. "Development/Maintenance/Reuse: Software Evolution in Product Lines". In: *Proceedings of the Conference on Software Product Lines : Experience and Research Directions*. Denver, Colorado, USA: Kluwer Academic Publishers, 2000, pp. 437–450.

[SV02] K. Schmid and M. Verlage. "The Economic Impact of Product Line Adoption and Evolution". In: *IEEE Software* 19.4 (July 2002), pp. 50–57.

[SV08] N. Szasz and P. Vilanova. "Statecharts and Variabilities". In: *International Workshop on Variability Modelling of Software-Intensive Systems*. 2008, pp. 131–140.

[TAB+15] L. Teixeira, V. Alves, P. Borba, and R. Gheyi. "A Product Line of Theories for Reasoning About Safe Evolution of Product Lines". In: *Proceedings of the International Conference on Software Product Lines*. SPLC '15. Nashville, Tennessee: ACM, 2015, pp. 161–170.

[TAK+14] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. "A Classification and Survey of Analysis Strategies for Software Product Lines". In: *ACM Comput. Surv.* 47.1 (June 2014), 6:1–6:45.

[TB07] A. K. Thurimella and B. Bruegge. "Evolution in Product Line Requirements Engineering: A Rationale Management Approach". In: *IEEE International Requirements Engineering Conference*. Oct. 2007, pp. 254–257.

[tBdVW17] M. H. ter Beek, E. P. de Vink, and T. A. C. Willemse. "Family-Based Model Checking with mCRL2". In: *Fundamental Approaches to Software Engineering*. Ed. by M. Huisman and J. Rubin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 387–405.

[TBK09] T. Thum, D. Batory, and C. Kastner. "Reasoning About Edits to Feature Models". In: *Proceedings of the International Conference on Software Engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 254–264.

[Tip95] F. Tip. "A Survey of Program Slicing Techniques". In: *Journal of Programming Languages* 3 (1995), pp. 121–189.

[TKE+11] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund. "Abstract Features in Feature Modeling". In: *International Software Product Line Conference*. Aug. 2011, pp. 191–200.

[TKH+12] L. Tahat, B. Korel, M. Harman, and H. Ural. "Regression Test Suite Prioritization Using System Models". In: *Software Testing, Verification and Reliability* 22.7 (2012), pp. 481–506.

[TKK+17] L. Tahat, B. Korel, G. Koutsogiannakis, and N. Almasri. "State-Based Models in Regression Test Suite Prioritization". In: *Software Quality Journal* 25.3 (Sept. 2017), pp. 703–742.

[TKP+18] G. Taentzer, T. Kehrer, C. Pietsch, and U. Kelter. "A Formal Framework for Incremental Model Slicing". In: *Fundamental Approaches to Software Engineering*. Ed. by A. Russo and A. Schürr. Cham: Springer International Publishing, 2018, pp. 3–20.

[TLS+10] C. Tao, B. Li, X. Sun, and C. Zhang. "An Approach to Regression Test Selection Based on Hierarchical Slicing Technique". In: *IEEE Annual Computer Software and Applications Conference Workshops*. July 2010, pp. 347–352.

[TM14] L. M. S. Tran and F. Massacci. "An Approach for Decision Support on the Uncertainty in Feature Model Evolution". In: *IEEE International Requirements Engineering Conference*. Aug. 2014, pp. 93–102.

[TTK04] A. Tevanlinna, J. Taina, and R. Kauppinen. "Product Family Testing: A Survey". In: *ACM SIGSOFT Software Engineering Notes* 29 (2004).

[UGK+08] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. "Testing Software Product Lines Using Incremental Test Generation". In: *International Symposium on Software Reliability Engineering*. Nov. 2008, pp. 249–258.

[UL06] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2006.

[UPL12] M. Utting, A. Pretschner, and B. Legeard. "A Taxonomy of Model-Based Testing Approaches". In: *Software Testing, Verification and Reliability* 22.5 (2012), pp. 297–312.

[UY13] H. Ural and H. Yenigün. "Regression Test Suite Selection Using Dependence Analysis". In: *Journal of Software: Evolution and Process* 25.7 (2013), pp. 681–709.

[VAT+18] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer. "A Classification of Product Sampling for Software Product Lines". In: *Proceedings of the International Systems and Software Product Line Conference*. SPLC '18. Gothenburg, Sweden: ACM, 2018, pp. 1–13.

[VBM15] M. Varshosaz, H. Beohar, and M. R. Mousavi. "Delta-Oriented FSM-Based Testing". In: *Formal Methods and Software Engineering*. Ed. by M. Butler, S. Conchon, and F. Zaïdi. Cham: Springer International Publishing, 2015, pp. 366–381.

[vdBee94] M. von der Beeck. "A Comparison of Statecharts Variants". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by H. Langmaack, W.-P. de Roever, and J. Vytopil. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 128–148.

[VSB+06] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[Weh05] H. Wehrheim. "Slicing Techniques for Verification Re-Use". In: *Theoretical Computer Science* 343.3 (2005). Formal Methods for Components and Objects, pp. 509–528.

[Weh06] H. Wehrheim. "Incremental Slicing". In: *Formal Methods and Software Engineering*. Ed. by Z. Liu and J. He. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 514–528.

[Wei08] D. M. Weiss. "The Product Line Hall of Fame". In: *International Software Product Line Conference*. Sept. 2008, pp. 395–395. URL: `http://splc.net/hall-of-fame/`.

[Wei10] S. Weißleder. "Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines". PhD thesis. Humboldt University of Berlin, 2010.

[Wei81] M. Weiser. "Program Slicing". In: *Proceedings of the International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449.

[Wei82] M. Weiser. "Programmers Use Slices when Debugging". In: *ACM Communication* 25.7 (July 1982), pp. 446–452.

[Wei84] M. Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10.4 (July 1984), pp. 352–357.

[WGA+13] S. Wang, A. Gotlieb, S. Ali, and M. Liaaen. "Automated Test Case Selection Using Feature Model: An Industrial Case Study". In: *Model-Driven Engineering Languages and Systems*. Ed. by A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 237–253.

[WHH03] C. Wohlin, M. Höst, and K. Henningsson. "Empirical Research Methods in Software Engineering". In: *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*. Ed. by R. Conradi and A. I. Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 7–23.

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.

[WRS+17] D. Wille, T. Runge, C. Seidl, and S. Schulze. "Extractive Software Product Line Engineering Using Model-based Delta Module Generation". In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. VAMOS '17. Eindhoven, Netherlands: ACM, 2017, pp. 36–43.

[WSS08] S. Weißleder, D. Sokenou, and B.-H. Schlingloff. "Reusing State Machines for Automatic Test Generation in Product Lines". In: *Workshop on Model-Based Testing in Practice*. 2008, p. 19.

[XQZ+05] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. "A Brief Survey of Program Slicing". In: *SIGSOFT Software Engineering Notes* 30.2 (2005), pp. 1–36.

[YH12] S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey". In: *Softw. Test. Verif. Reliab.* 22.2 (2012), pp. 67–120.

[YM12] A. R. Yazdanshenas and L. Moonen. "Fine-Grained Change Impact Analysis for Component-Based Product Families". In: *IEEE International Conference on Software Maintenance*. Sept. 2012, pp. 119–128.

[Zav93] P. Zave. "Feature Interactions and Formal Specifications in Telecommunications". In: *Computer* 26.8 (Aug. 1993), pp. 20–28.

[Zha02] J. Zhao. "Change Impact Analysis for Aspect-Oriented Software Evolution". In: *Proceedings of the International Workshop on Principles of Software Evolution*. IWPSE '02. Orlando, Florida: ACM, 2002, pp. 108–112.

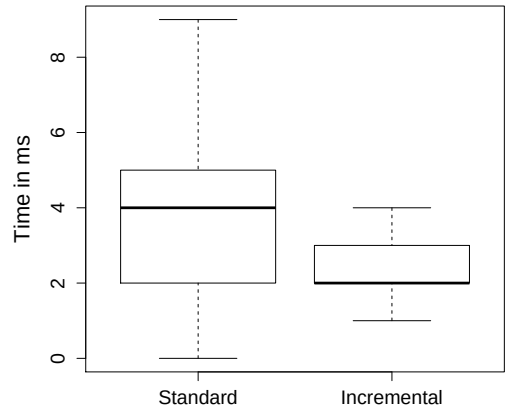
[ZYX+02] J. Zhao, H. Yang, L. Xiang, and B. Xu. "Change Impact Analysis to Support Architectural Evolution". In: *Journal of Software Maintenance and Evolution: Research and Practice* 14.5 (2002), pp. 317–333.

A Additional Results of the Incremental Slicing Evaluation

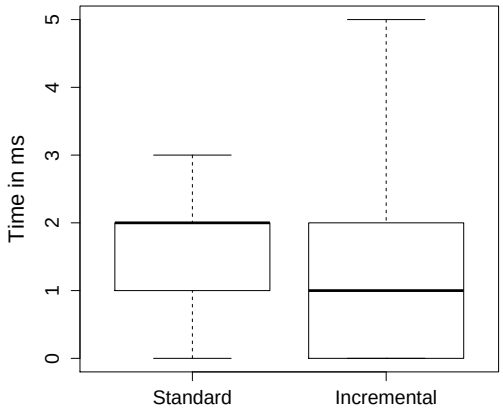
In this section, we provide further results of the evaluation of our slicing-based change impact analysis technique. In Tab. A.1, the percentages of outliers w.r.t. the complete number of executions are summarized. In Fig. A.1, the box plots for the dependency analysis as well as slicing runtime comparison for the versions θ_1 and θ_3 of the Wiper SPL are shown. In Fig. A.2, the box plots for the dependency analysis as well as slicing runtime comparison for the versions θ_1, θ_2, and θ_3 of the Vending Machine SPL are depicted and in Fig. A.3 for the version θ_5. In Tab. A.2, the results of the hypothesis test regarding the runtime comparisons are summarized.

Table A.1: Percentage of Runtime Outliers

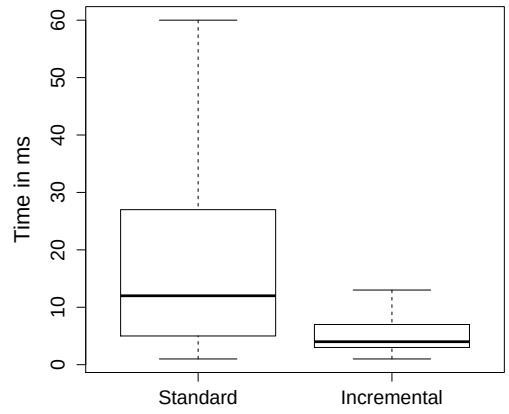
SPL	#Executions	Dependency Analysis		Slicing	
		Standard	Incremental	Standard	Incremental
W_{θ_0}	2,800	2.50	2.35	2.50	0.75
W_{θ_1}	2,800	0.60	6.03	6.89	0.00
W_{θ_2}	6,600	1.21	9.78	1.71	1.06
W_{θ_3}	27,600	10.58	6.03	3.09	0.85
W_{θ_4}	27,600	6.66	9.84	3.26	1.97
VM_{θ_0}	37,800	1.65	0.36	0.88	3.33
VM_{θ_1}	86,100	0.80	2.13	8.99	0.17
VM_{θ_2}	241,500	0.37	0.81	4.35	4.13
VM_{θ_3}	86,100	0.79	1.02	3.29	3.46
VM_{θ_4}	86,100	0.43	0.45	0.73	0.22
VM_{θ_5}	86,100	0.36	1.86	8.07	0.30
VM_{θ_6}	112,800	8.64	3.11	0.73	5.37
MP_{θ_0}	12,000	15.03	5.06	3.72	1.93
MP_{θ_1}	1,2000	0.00	11.18	2.63	1.58
MP_{θ_2}	49,600	0.15	6.84	13.82	0.56



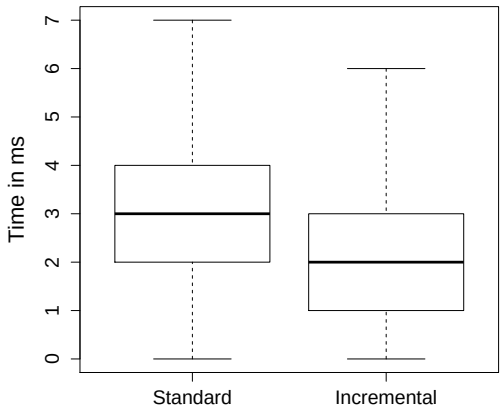
(a) Dependency Analysis Runtime for Wiper SPL Version θ_1



(b) Slicing Runtime for Wiper SPL Version θ_1

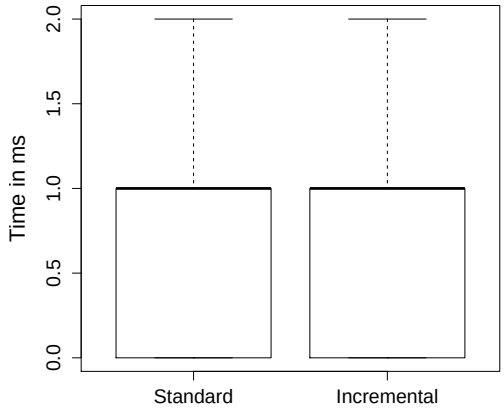


(c) Dependency Analysis Runtime for Wiper SPL Version θ_3

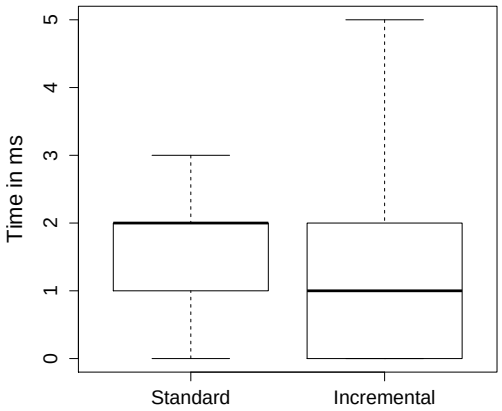


(d) Slicing Runtime for Wiper SPL Version θ_3

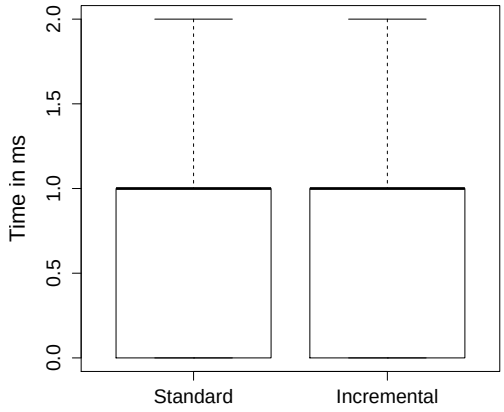
Figure A.1: Runtime Results of the Dependency Analysis and Slice Computation for the Versions θ_1 and θ_3 of the Wiper SPL



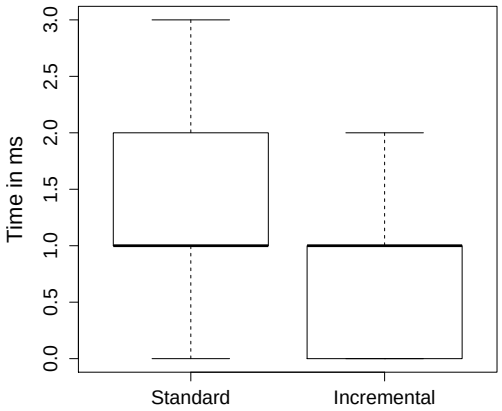
(a) Dependency Analysis Runtime for Vending Machine SPL Version θ_1



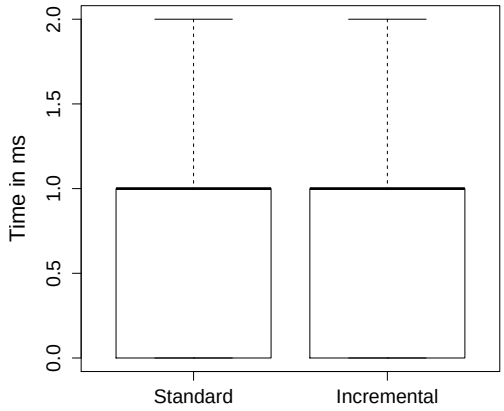
(b) Slicing Runtime for Vending Machine SPL Version θ_1



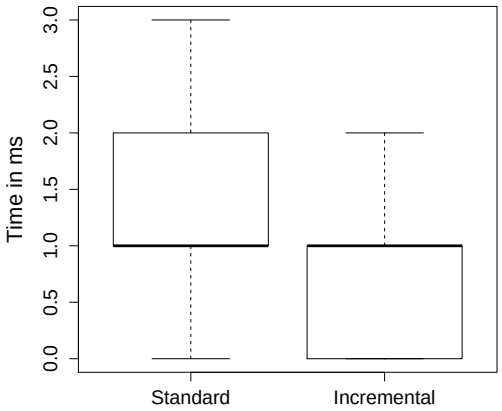
(c) Dependency Analysis Runtime for Vending Machine SPL Version θ_2



(d) Slicing Runtime for Vending Machine SPL Version θ_2

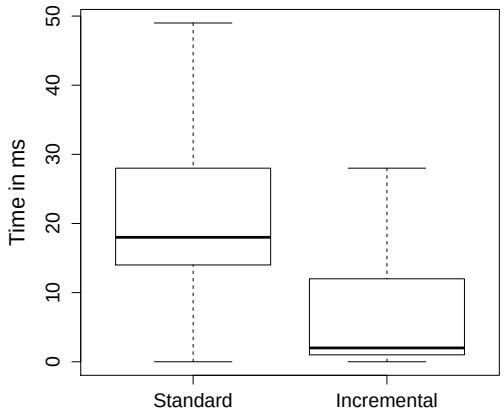


(e) Dependency Analysis Runtime for Vending Machine SPL Version θ_3

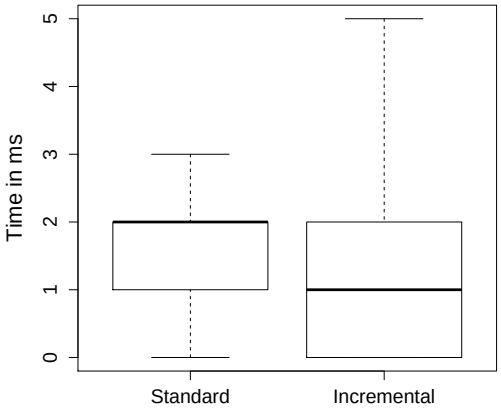


(f) Slicing Runtime for Vending Machine SPL Version θ_3

Figure A.2: Runtime Results of the Dependency Analysis and Slice Computation for the Versions θ_1, θ_2, and θ_3 of the Vending Machine SPL



(a) Dependency Analysis Runtime for Vending Machine SPL Version θ_5



(b) Slicing Runtime for Vending Machine SPL Version θ_5

Figure A.3: Runtime Results of the Dependency Analysis and Slice Computation for the Version θ_5 of the Vending Machine SPL

Table A.2: Results (p-Values) of One-Sided Hypothesis Test of Runtime Comparisons Using Wilcoxon-Mann-Whitney-Test

SPL	Dependency Analysis	Slicing
	$H_1 : (\mu_{Std} - \mu_{Incr}) > 0$	$H_1 : (\mu_{Std} - \mu_{Incr}) > 0$
W_{θ_0}	1	$2.2e^{-16}$
W_{θ_1}	$2.2e^{-16}$	$2.2e^{-16}$
W_{θ_2}	$2.2e^{-16}$	$2.2e^{-16}$
W_{θ_3}	$2.2e^{-16}$	$2.2e^{-16}$
W_{θ_4}	$2.2e^{-16}$	$2.2e^{-16}$
VM_{θ_0}	$2.2e^{-16}$	$2.2e^{-16}$
VM_{θ_1}	1	$2.2e^{-16}$
VM_{θ_2}	$2.2e^{-16}$	$2.2e^{-16}$
VM_{θ_3}	1	$2.2e^{-16}$
VM_{θ_4}	$2.2e^{-16}$	$2.2e^{-16}$
VM_{θ_5}	$2.2e^{-16}$	$2.2e^{-16}$
VM_{θ_6}	$2.2e^{-16}$	$2.2e^{-16}$
MP_{θ_0}	1	$2.2e^{-16}$
MP_{θ_1}	$2.2e^{-16}$	$2.2e^{-16}$
MP_{θ_2}	$2.2e^{-16}$	$2.2e^{-16}$

Publications

This thesis is based on the following peer-reviewed publications.

[LNT+19] **S. Lity**, M. Nieke, T. Thüm, and I. Schaefer. Retest Test Selection for Product-Line Regression Testing of Variants and Versions of Variants. In *Journal of Systems and Software*, volume 147, pages 46–63, 2019.

[LAT+17] **S. Lity**, M. Al-Hajjaji, T. Thüm, and I. Schaefer. Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-line Analysis. In *International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS 2017, pages 60–67, 2017.

[LKS16] **S. Lity**, M. Kowal, and I. Schaefer. Higher-Order Delta Modeling for Software Product Line Evolution. In *International Workshop on Feature-Oriented Software Development*, FOSD 2016, pages 39–48, 2016.

[LMT+16] **S. Lity**, T. Morbach, T. Thüm, and I. Schaefer. Applying Incremental Model Slicing to Product-Line Regression Testing. In *International Conference on Software Reuse: Bridging with Social-Awareness*, ICSR 2016, pages 3–19, 2016.

[LBS15] **S. Lity**, H. Baller, and I. Schaefer. Towards Incremental Model Slicing for Delta-Oriented Software Product Lines. In *International Conference on Software Analysis, Evolution, and Reengineering*, SANER (ERA Track) 2015, pages 530-534, 2015.

[LLS+12] **S. Lity**, M. Lochau, I. Schaefer, and U. Goltz. Delta-Oriented Model-Based SPL Regression Testing. In *International Workshop on Product LinE Approaches in Software Engineering*, PLEASE 2012, pages 53-56, 2012.

Further peer-reviewed publications related to this thesis.

[LRB+19] M. Lochau, D. Reuling, J. Bürdek, T. Kehrer, **S. Lity**, A. Schürr, and U. Kelter. Model-Based Round-Trip Engineering and Testing of Evolving Software Product Lines. In *Managed Software Evolution*, pages 141–173, Springer, 2019.

[LNT+18] **S. Lity**, S. Nahrendorf, T. Thüm, C. Seidl, and I. Schaefer. 175% Modeling for Product-Line Evolution of Domain Artifacts. In *International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS 2018, pages 27–34, 2018.

[RLB+18] S. Ruland, L. Luthmann, J. Bürdek, **S. Lity**, T. Thüm, M. Lochau, and M. Ribeiro. Measuring Effectiveness of Sample-Based Product-Line Testing. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, pages 119–133, 2018.

[LBL+17] R. Lachmann, S. Beddig, **S. Lity**, S. Schulze, and I. Schaefer. Risk-Based Integration Testing of Software Product Lines. In *International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS 2017, pages 52–59, 2017.

[ALL+17] M. Al-Hajjaji, **S. Lity**, R. Lachmann, T. Thüm, I. Schaefer, and G. Saake. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *International Workshop on Variability and Complexity in Software Design*, VACE 2017, pages 34-40, 2017.

[LLA+16] R. Lachmann, **S. Lity**, M. Al-Hajjaji, F. Fürchtegott, and I. Schaefer. Fine-Grained Test Case Prioritization for Integration Testing of Delta-Oriented Software Product Lines. In *International Workshop on Feature-Oriented Software Development*, FOSD 2016, pages 1–10, 2016.

[LBL+15] **S. Lity**, J. Bürdek, M. Lochau, M. Berens, A. Schürr, and I. Schaefer. Re-Engineering Automation Systems as Dynamic Software Product Lines. In *Dagstuhl-Workshop Model-Based Development of Embedded Systems*, MBEES 2015, 2015.

[LFH+15] J. Ladiges, A. Fay, C. Haubeck, W. Lammersdorf, **S. Lity**, and I. Schaefer. Supporting Commissioning of Production Plants by Model-Based Testing and Model Learning. In *International Symposium on Industrial Electronics*, ISIE 2015, pages 606-611, 2015.

[LLL+15] R. Lachmann, **S. Lity**, S. Lischke, S. Beddig, S. Schulze, and I. Schaefer. Delta-Oriented Test Case Prioritization for Integration Testing of Software Product Lines. In *International Software Product Line Conference*, SPLC 2015, pages 81–90, 2015.

[BLL+14] H. Baller, **S. Lity**, M. Lochau, and I. Schaefer. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *International Conference on Software Testing, Verification and Validation*, ICST 2014, pages 303-312, 2014.

[LLL+14] M. Lochau, **S. Lity**, R. Lachmann, I. Schaefer, and U. Goltz. Delta-Oriented Model-Based Integration Testing of Large-Scale Systems. In *Journal of Systems and Software*, volume 91, pages 63–84, 2014.

[LBL+14] M. Lochau, J. Bürdek, **S. Lity**, M. Hagner, C. Legat, U. Goltz, and A. Schürr. Applying Model-Based Software Product Line Testing Approaches to the Automation Engineering Domain. In *at-Automatisierungstechnik*, volume 62, pages 771–780, 2014.

[LSK+12] M. Lochau, I. Schaefer, J. Kamischke, and **S. Lity**. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *Tests and Proofs*, TAP 2012, pages 67–82, 2012.

Technische Universität Carolo-Wilhelmina zu Braunschweig (Germany)
Institute of Software Engineering and Automotive Informatics

Mühlenpfordtstr. 23
D-38106 Braunschweig